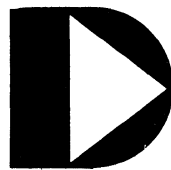


# **TIME SHARING DATABUS DATASHAR 3.1 Users's Guide**

October 14, 1974

Model Code No. 50048

**DATAPPOINT CORPORATION**



**The Leader in  
Dispersed Data Processing**

DATASHARE 3 USER'S GUIDE

November 1974

DATAPoint CORPORATION

## TABLE OF CONTENTS

	PAGE
1. INTRODUCTION	1
2. STATEMENT STRUCTURES	
2.1 Comments	6
2.2 Compiler directives	6
2.3 File declaration and data definition	6
2.4 Program execution	7
2.5 Literals	8
2.6 The forcing character	8
2.7 A sample program	10
3. FILE DECLARATION AND DATA DEFINITION	
3.1 File declaration	11
3.2 Data definition	11
3.2.1 Numeric string variables	13
3.2.2 Character string variables	13
3.2.3 Common Data Areas	14
4. PROGRAM CONTROL INSTRUCTIONS	
4.1 GOTO	15
4.2 BRANCH	15
4.3 CALL	16
4.4 RETURN	16
4.5 STOP	16
4.6 CHAIN	17
4.7 TRAP	18
4.8 TRAPCLR	20
4.9 ROLLOUT	21
4.10 PI	23
4.11 TABPAGE	24
5. CHARACTER STRING HANDLING INSTRUCTIONS	
5.1 MOVE	25
5.2 APPEND	27
5.3 MATCH	28
5.4 CMOVE	29
5.5 CMATCH	29
5.6 BUMP	29
5.7 RESET	30
5.8 ENDSET	31
5.9 LENSET	31
5.10 CLEAR	31
5.11 EXTEND	32
5.12 LOAD	32
5.13 STORE	32
5.14 CLOCK	33
5.15 TYPE	34

6. ARITHMETIC INSTRUCTIONS	
6.1 ADD	36
6.2 SUB	36
6.3 MULT	36
6.4 DIV	36
6.5 MOVE	37
6.6 COMPARE	37
6.7 LOAD	37
6.8 STORE	37
7. INPUT/OUTPUT INSTRUCTIONS	
7.1 KEYIN	39
7.2 DISPLAY	43
7.3 CONSOLE	43
7.4 BEEP	44
7.5 PRINT	44
7.6 RELEASE	46
7.7 Disk I/O	47
7.7.1 File structures	47
7.7.2 Positioning and accessing	50
7.7.3 PREPARE	55
7.7.4 OPEN	57
7.7.5 CLOSE	58
7.7.6 READ	58
7.7.7 READKS	66
7.7.8 WRITE	67
7.7.9 WRITAB	70
7.7.10 UPDATE	71
7.7.11 INSERT	72
7.7.12 DELETE	72
7.7.13 WEOF	73
8. PROGRAM GENERATION	
8.1 Preparing source files	75
8.2 Compiling source files	75
8.3 Compilation directives	79
8.4 Compilation diagnostics	80
8.5 Disk space requirements	81
9. SYSTEM GENERATION	
9.1 Loading from cassette	82
9.2 System configuration	82
9.3 Necessary programs	83
10. SYSTEM OPERATION	
10.1 Bringing up the system	84
10.2 Taking down the system	87
10.3 Fatal error conditions	87
11. ANSWER AND MASTER CONCEPTS	
11.1 System security	88
11.2 System convenience	88
11.3 Sample Answer and Master programs	89



12. PHYSICAL SYSTEM CHARACTERISTICS	
12.1 Virtual Memory	92
12.2 Major Modules	94
12.3 Scheduling	97
13. PHYSICAL INSTALLATION	
13.1 Main Peripherals	99
13.2 Terminal Connections	100
13.3 Port Speed Selection	103
13.4 Non-3360-102 Terminal Device	104

#### APPENDICES

- A. Instruction summary
- B. I/O list controls
- C. ANSWER and MASTER program examples
- D. File access lockout program example
- E. Compiler error codes
- F. Interpreter error codes

## 1. INTRODUCTION

DATASHARE permits the simultaneous execution of up to eight DATABUS programs, each dealing with its own remote Datapoint CRT terminal (a system option allows one of the programs to execute using the system console instead of a remote terminal which allows DATASHARE to be run without a multi-port adaptor). The DATASHARE interpreter runs under the Disk Operating System (taking advantage of all of its file handling characteristics), handles a high-speed line printer or servo printer, provides indexed-sequential as well as random and sequential file accessing, and allows intra-file access, thus providing a powerful data entry and processing facility. This configuration allows a flexible mix of remote, batch, and interactive processing all under the control of a high level language program, enabling the user to configure the system to best suit his data processing needs.

In addition, the DOS with its variety of utility and higher level language systems may be used alternately to DATASHARE, enabling processing of tasks not appropriate to the multiple terminal environment.

Using virtual memory techniques, DATASHARE provides each program with a 16K byte area for executable statements. This, in combination with the ability of the compiler to accommodate over 3400 labels, enables the user to create and use programs of over one hundred pages (a very large high level language program). To provide rapid program execution, the data area for each program is maintained in main memory and not swapped. A combined total of 4096 bytes of main memory is allocated for the combined data area of all ports configured into the system. The system can be configured to run with one through eight ports with the data area being variably partitioned among them (the data area of any one port can be configured to be from 20 bytes to almost all of the total area available). If the system is configured to run one of the ports on the console, 3759 bytes of data area are available to the ports configured. If the system is configured to run the servo printer as the system printer, 3584 bytes of data area are available to the ports configured. If the system is configured to run one of the ports on the console and run the servo printer as the system printer, 3248 bytes of data area are available to the ports configured.

Any of the Datapoint 2200 printer systems may be connected to the DATASHARE configuration with printing being controlled from any of the ports. If the printer is busy with one port, another port trying to access the printer will wait until the first port releases the printer.

All program execution in DATASHARE occurs in the DATABUS language. Terminal command interpretation is handled in special ANSWER and MASTER programs (unique for each port) which also handle system security. These programs are provided with the system but may be compiled like any other Databus program, enabling the user to completely define his own terminal command and security system.

Program generation is performed under the DOS using the general purpose DOS editor and DATASHARE compiler.

## N O T E

DATASHARE 3 has the following new features over DATASHARE 2. It will be noted where these features can cause minor incompatibilities with programs written for DATASHARE 1 or DATASHARE 2.

1. In addition to physically random and sequential file access, indexed file access is available (see Section 7.7). Indexed files can be accessed either directly, based on a key value, or sequentially by the collating sequence of the key values, as well as still being accessed physically (key values within a single index may not be duplicated). Any number of indices can be created and maintained for a given data file. The indices are separate files of pointers into the data file. This enables expansion capability of the index without overflow problems. It also means that the data file is still fully compatible with all existing software since it contains no special data structuring. Lack of structuring in the data file also implies greater recoverability of data in the event of file system failure. The indexed-sequential feature does impose a slight incompatibility with previous versions of DATASHARE. To allow for the greater amount of information that must be stored for an open file, the logical file information has been moved out of the interpreter working storage into the user's data area. This means that all logical files must be declared in the user's data area which will lessen the amount of data area left for variables used within the user's program (see Sections 2.3 and 3.1). This lessening is abated somewhat by the availability of literals in certain DATASHARE operations. An advantage of the declaration of logical files in the user's data area is that the user may now have any number of logical files active at any one time, the number being limited only by the amount of data area that the user's has available in a given program. The declaration of logical files will require the user to add the file declaration statements at the beginning of his program and change all disk I/O statements to reference the declaration names instead of the simple digits supplied in

DATASHARE 2. However, this conversion is mechanically achievable since he need only compile his original program with the new DATASHARE 3 compiler and it will flag all statements which need to be changed.

2. Literals may now be supplied in CHAIN ROLLOUT STORE MOVE APPEND MATCH ADD SUBTRACT MULTIPLY DIVIDE COMPARE OPEN and PREPARE statements alleviating the need for fixed constant data in the user's data area (see Section 2.5).

3. The data area can be partitioned in any way among the ports instead of being forced to be equally shared among all ports configured to run (see Section 9.2).

4. The system can be configured to execute port one on the system console instead of on the terminal connected to the first port of the multi-port adaptor. The system console has all the capabilities of a 3360-102 except that only the first 12 lines are available. This can either save one terminal station or it can be used to allow DATASHARE programs to be executed on systems without a multi-port adaptor (see Section 9.2). If port one is being run on the system console, an additional option allows the rest of the ports to execute (as long as a KEYIN statement is not executed) even though a multi-port adaptor is not connected to the system. This allows such tasks as print file spooling to be performed while other processing is active on port one.

5. The servo printer can be configured to be used as the system printer instead of one of the standard line printers (see Sections 7.5 and 9.2).

6. Echo during input operations from the terminal can be defeated, enhancing the message switching capability of the system (see Section 7.1).

7. A pause operator in KEYIN and DISPLAY statements has been added which allows a program to pause for any number of seconds with very little system overhead (see Section 7.1).

8. A new arithmetic package has been incorporated into the DATASHARE interpreter. This package performs division without the previous restrictions upon the number of places after a decimal point (e.g., performs division the way one would expect it to be performed) but does, however, have reasonable restrictions upon the length of operands used in multiplication and division (see Section 6.3 and 6.4). These restrictions could, however, cause a program to operate somewhat differently when doing multiplication or division, so these operations should be checked carefully when converting from one of the previous versions of DATASHARE.

9. The APPEND instruction will append a numeric item into a character string item. The operation is performed as if the numeric item was a string with a formpointer pointing to its first character and a logical length pointing to its last character (see Section 5.2).

10. The Julian date is incremented when the clock changes from 23:59:59 to 00:00:00. However, the date is not checked for the end of the year, so the date must be changed manually at midnight of the end of the year.

11. The extension of DATASHARE 3 object files has been changed to DS3. The new DATASHARE 3 interpreter will not run object code generated by the DATASHARE 1 or DATASHARE 2 compilers.

12. The DATASHARE 3 compiler has been greatly enhanced in its label capability. Due to the fact that it keeps its label dictionary on the disk, the compiler can now handle over 3400 labels. The compiler also has the ability to list referenced lines at the right margin of 132 column printers. Additional compiler features include more illuminating syntax error messages, the EQU and INCLUDE directives, and the ability to generate a listing on the servo printer as well as the standard line printers.

13. The DATASHARE interpreter may be named any name as long as the command (/CMD) file and all six of the overlays have the same name. For example, if DS/CMD was renamed ABC/CMD, then DS/OV1 would have to be renamed ABC/OV1, DS/OV2 would have to be renamed ABC/OV2, etc. The command file may no longer be specified in the DOS AUTO command. However, a new program (called AUTOKEY) has been written which allows any DOS command to be automatically executed (see Section 10.1).

14. The I/O error trap has been augmented by an additional letter which indicates the nature of the problem. This will be of great help in debugging programs as there are now 19 different types of I/O errors. The different types of I/O errors cannot be differentiated if the trap is set, but if the trap is not set the different letters will aid the programmer in determining what caused the problem.

15. If port one is configured to run on the system console, a name may be specified on the DOS command after the name of the DATASHARE interpreter. This name is the name of the ANSWER program used by the port one program (it must always be on drive zero). Therefore, if the DATASHARE system is being used to simply run batch processing programs, the CHAIN utility can be used to execute DATASHARE processing functions (see Section 10.1).

16. Logical files can no longer be keep open through a CHAIN operation. This implies that all files should be closed before a CHAIN is performed to the next program. It also implies that if new space was allocated in a file and then the interrupt key is struck, none of the space will be deallocated. (It is now useless to have the ANSWER and MASTER programs contain CLOSE statements, implying that these programs must be changed as well as all other file accessing programs.)

17. An instruction has been added which enables the programmer to prevent background execution interruptions (from his time running out, a higher priority process wanting to run, or from the INT key on the port terminal) for up to 20 Databus instruction executions. This enables common file access conflicts to be resolved in a very efficient way and also allows the programmer to prevent the operator from destroying a file structure by accidentally striking the INT key at the wrong time. The limit of 20 instruction executions prevents the program from accidentally hanging up the system background execution. Note that even though background interruptions are prevented, the foreground one millisecond interrupt driven port handling routines are still executing meaning other ports can still be entering or displaying data through the use of the port I/O statements.

18. The DSBACK command has been improved to display the correct number of ports (the screen is blanked if port one is executing on the console). Also, a new command has been added (called DSBACKTD) which allows the system operator to initialize the time and date before execution is resumed.

19. A READ operation whose list is terminated by a semicolon will now detect an end of file mark. This change could require a change in the program logic in DATASHARE 2 programs.

## 2. STATEMENT STRUCTURES

There are five basic types of statements in DATASHARE: comment, compiler directive, file declaration, data definition, and program execution.

### 2.1 Comments

Comment lines have a period, asterisk, or plus sign in the first column, and may appear anywhere in the program. Comments are most useful in explaining program logic and subroutine function and parameterization to enable someone reading through the program to more easily understand its logic. The comment which begins with an asterisk will be printed at the top of the next page if fewer than 12 lines are available at the bottom of the current page. This allows comments to be presented on the same page as the program statements without having to know where the listing currently stands on the page. The comment which begins with a plus sign will always be printed at the top of the next page. This allows major sections of the program to be started at the top of a page. Use of the asterisk at the beginning of each section or subroutine description is encouraged since this greatly enhances program readability. Use of the plus sign should be cautious since it can easily waste great quantities of paper.

### 2.2 Compiler directives

Compiler directives enable the programmer to include other files in the current compilation and to define the absolute value of a symbolic name for use in tab positioning in file I/O statements and column positioning in terminal I/O statements. The inclusion directive allows one to break a large program into several files for ease in editing. Another useful aspect is that one can have a common set of subroutines or data definition blocks which are included into a number of different programs. Therefore, when a change is made in one of the routines or in the definition of a data item, one need edit the change only once, reducing both the amount of manual labor involved and the chance for error. See Section 8 (Program Generation) for more complete information on the use of compiler directives.

### 2.3 File declarations and data definitions

File declaration and data definition statements must occur before any program execution statements and are used for setting up all of the logical files and data variables in the program. All file declaration and data definition statements must have labels. All compiler directive, file declaration, and data definition statement labels must be unique among themselves. Program execution statements must

appear after any file declaration or data definition statements and may or may not have labels. The labels on program execution statements may be the same as labels on the compiler directive, file declaration, and data definition statements. Program execution always begins with the first executable statement.

#### 2.4 Program execution

Labels for variables and executable statements can consist of any combination of up to eight letters and digits beginning with a letter. The following are examples of valid labels:

```
A
ABC
A1BC
B1234
ABCDEF
BIGLABEL
```

The following are examples of invalid labels:

```
HI,JK (contains an invalid character)
4DOGS (does not begin with a letter)
```

Statements other than comments consist of a label field, an operation field, an operand field, and a comment field. The label field is considered empty if a space appears in the first column of the line. The operation field denotes the operation to be performed on the following operands. In many operations, two operands are required in the operand field. These operands may be connected either by an appropriate preposition (BY, TO, OF, FROM, or INTO) or a comma. One or more spaces should follow each element in a statement except where a comma is used, in which case the comma must be the terminating character of the previous element and may be followed by any number (including zero) of spaces. For example, the following are all examples of valid statements:

```
LABEL1 ADD PCS TO TOTAL
LABEL2 ADD PCS OF TOTAL THIS IS A COMMENT
LABEL3 ADD PCS, TOTAL
LABEL4 ADD PCS, TOTAL
```

Note that any preposition may be used even if it does not make sense in English. The following are examples of invalid statements:

```
LABEL1 ADD PCS TOTAL (missing connective)
LABEL2 ADD PCS ,TOTAL (space before comma)
```



Certain DATASHARE statements allow a list of items to follow the operation field. In many cases, this list can be longer than a single line, in which case the line must be continued. This is accomplished by replacing the comma that would normally appear in the list with a colon and continuing the list on the following line. For example, the two statements:

```

DISPLAY A,B,C,D:
          E,F,G
DISPLAY A,B,C,D,E,F,G

```

will perform the same function.

## 2.5 Literals

In an effort to reduce the amount of data area needed by a program, literals are allowed in certain statements which would otherwise need constant data in the user's data area. The instructions which can contain literals are: STORE, ROLLOUT, CHAIN, MOVE, APPEND, MATCH, ADD, SUB, MULT, DIV, COMPARE, OPEN, and PREPARE. In all except the program control and I/O statements, the literal must be the first operand. The literal is always enclosed within a pair of double quotes (see the following section on the forcing character) and may be from 1 through 40 characters in length (excluding the quotes). When a literal is used as a string variable, its formpointer is always equal to one and its logical length always points to the last character that is quoted. Examples of the statements which can contain literals follow:

```

STORE      "APPLES" INTO X OF S1,S2,S3
ROLLOUT    "CHAIN FIX22"
CHAIN      "NEXTPROG"
OPEN       FILE1,"DATAFILE"
PREPARE    FILE1,"USERDATA"
MOVE       "MESSAGE" TO M3442
MOVE       "100.55" TO VALUE
APPEND     "." TO STR1
MATCH      "YES" TO ANSWER
ADD        "23.46" TO TOTAL
SUB        "1" FROM COUNT
MULT       ".1" BY TAX
DIV        "33.3333" INTO FACTOR
COMPARE    "10" TO LINENUMB

```

## 2.6 The forcing character

The pound sign (#) is interpreted by the compiler as a forcing character in any quoted item which can contain multiple characters. The character immediately following the pound sign is used in the quoted item simply as a

character value regardless of its significance to the compiler. Thus, the pound sign itself and the quote (") may be used in DATASHARE statements. For example,

```
DISPLAY "CUSTOMER## SHOULD BE #"2222#" "
```

would display exactly:

```
CUSTOMER# SHOULD BE "2222"
```

on the screen. Note carefully the wording used above to describe the cases where the pound sign is used to denote a forcing character. This wording excludes the cases of RESET, CMATCH, and CMOVE since those operations cannot have quoted items which contain multiple characters. For example,

```
CMOVE "" TO STRING
```

would be used to move a double quote sign into the variable STRING. However, the use of a literal in a MOVE instruction would require the use of the forcing character, even in a single character move, since the quoted item can be a multiple character quote. For example:

```
MOVE "#" TO STRING
```

would be used to move a double quote sign into the variable STRING. The RESET, CMOVE, and CMATCH instructions are the only exceptions to the forcing character convention within quoted items.

Examples:

```
RESET STRING TO "#"  
CMOVE "#" TO STRING  
CMATCH "" TO STRING
```

## 2.7 A sample program

```
. PROGRAM TO DISPLAY A MULTIPLICATION TABLE
.
COUNT1  FORM      "0"
COUNT2  FORM      "0"
PROD     FORM      2
*
. HERE IS THE START OF THE EXECUTABLE CODE
.
START    DISPLAY *ES,"MULTIPLICATION TABLE:",*N
LOOP     MOVE     COUNT1 TO PROD
         MULT     COUNT2 BY PROD
         DISPLAY  COUNT1,"X",COUNT2,"=",PROD," ";
         ADD      "1" TO COUNT2
         GOTO     LOOP IF NOT OVER
         DISPLAY *N
         ADD      "1" TO COUNT1
         GOTO     LOOP IF NOT OVER
         STOP
```

### 3. FILE DECLARATION AND DATA DEFINITION

There are two types of statements in DATASHARE which cause space within the user's data area to be assigned. The first is logical file declaration where the space is used to store the DOS system information about the file being used and the second is data definition where the space is used to keep the variable information within the DATASHARE program.

#### 3.1 File declaration

Two types of files can be declared in DATASHARE. The first is a type that will be used for random or physically sequential accessing. This type is declared using the FILE statement:

```
INFILE FILE
```

The label INFILE is the label which will be used in all disk I/O statements that are to use this particular logical file. This statement causes 17 bytes of data area to be consumed. This area stores the 15 bytes used in the DOS logical file table, a space compression counter, and a flag indicating that this is a physically random or sequential access only file. Note that since logical file information is stored in the user's data area, the user may have any number of logical files active at any one time providing his data area will contain all of the necessary declaration information.

The second type of file declaration is used for indexed-sequential file accessing. This type is declared using the IFILE statement:

```
ISAMFILE IFILE
```

The label ISAMFILE is the label which will be used in all disk I/O statements which are to use this particular logical file. This statement causes 26 bytes of data area to be consumed. This area stores the information that a FILE declaration stores plus three three-byte pointers for use in the access method. These pointers point to the beginning of the last record accessed (for updating operations), to the next sequential key (for sequential by key accessing), and to the DOS R.I.B. of the index file (used in all accessing operations).

#### 3.2 Data definition

There are two types of data used within the DATASHARE language. They are numeric strings and character strings. The arithmetic operations are performed on numeric strings and string operations are performed on character strings. There are also operations allowing movement of numeric

strings into character strings and vice versa. Numeric strings have the following memory format:

```
0200      1      2      .      3      0203
```

The leading character (0200) is used as an indicator that the string is numeric. The trailing character (0203) is used to indicate the location of the end of the string (ETX). Note that the format of a numeric string is set at definition time and does not change throughout the execution of the program. A numeric string can be defined to contain at most 22 characters.

When a move into a number occurs from a string or differently formatted number, reformatting will occur to cause the information to assume the format of the destination number (decimal point position and the number of digits before and after the decimal point) with truncation occurring if necessary (rounding occurs if truncation is to the right of the decimal point).

Character strings have the following memory format:

```
9      5      THE QUICK BROWN      0203
```

The first byte is called the logical length and points to the last character currently being used in the string (K in the above example). The second byte is called the formpointer and points to a character currently being used in the string (Q in the above example). The use of the logical length and formpointer in character strings will be explained in more detail in the explanations of each character string handling instruction. Basically, however, these pointers are the mechanism via which the programmer deals with individual characters within the string.

The term physical length will be used to mean the number of possible data characters in a string (15 in the above example). The logical and physical lengths of string variables is limited to 127.

Whenever a data variable is to be used in a program, it must be defined at the beginning by using either the FORM, INIT, or DIM instructions. These instructions reserve the memory space described above for the data variable whose name is given in the label field. Note that all variables must be defined before the first executable statement is given in the program and that once an executable statement

is given, no more variables may be defined. Numeric strings are created with the FORM instruction while character strings are created with the INIT or DIM instruction.

### 3.2.1 Numeric string variables

Numeric variables are defined in one of two ways with the FORM instruction as shown in the following illustration:

```
EMRATE  FORM 4.2
XAMT    FORM " 382.4 "
```

In this example, EMRATE has been defined as a string of decimal digits which can cover the range from 9999.99 to -999.99. The FORM instruction illustrated reserves spaces in memory for a number with four places to the left of a decimal point and two places to the right of a decimal point and initializes the value to zero. When the number is negative, one of the places to the left of the decimal point is used by the minus sign. XAMT, in the example, is defined with four places to the left of the decimal point and three to the right but with an initial decimal value of 382.400. The physical length of a numeric variable is limited to 22 characters (decimal point and sign included).

### 3.2.2 Character string variables

Character strings are defined with either a DIM or INIT instruction. DIM reserves a space in memory for the given number of characters but sets the logical length and formpointer to zero and initializes all the characters to spaces. For example:

```
STRING DIM 25
```

A character string can also be defined with some initial value by using the INIT instruction. For example:

```
TITLE  INIT "PAYROLL PROGRAM"
```

initializes the string TITLE to the characters shown and gives it a logical length of 15. The formpointer is set to one. Note that in the case of strings, the actual amount of memory space reserved is three bytes greater than the number specified in the DIM or quoted in the INIT instruction (TITLE occupies 18 bytes in memory, 15 of which hold characters).

Octal control characters (000 to 037) may be included when initializing a string. The control character is separated by commas, without quotes, and is preceded by a zero. For example,

TITLE INIT "PAYROLL PROGRAM",015,"TEST1"

would initialize a string with a logical and physical length of 21 characters. The octal control character, 015, would appear after the M in PROGRAM and before the first T in TEST1. This feature is included mainly for message switching applications and for allowing control of ASR Teletype compatible terminals. It is the responsibility of the programmer to remember that some of these characters (000, 003, 011, and 015) are used for control purposes in disk files.

### 3.2.3 Common data areas

Since DATASHARE has the provision to chain programs so that one program can cause another to be loaded and run, it is desirable to be able to carry common data variables from one program to the next. The procedure for doing this is as follows:

- a. Identify those variables to be used in successive programs and in each program define them in exactly the same order and way and preferably at the beginning of each program. The point in this is to cause each common variable to occupy the same locations in each program. Strange results in program execution usually occur if a common variable is misaligned with respect to the variable in the previous program.
- b. For the first program to use the variables, define them in the normal way. Then, for all succeeding programs, place an asterisk in each FORM, DIM, or INIT statement, as illustrated below, to prevent those variables from being initialized when the program is loaded into memory.

Examples:

```
MIKE FORM *4.2
JOE DIM *20
BOB INIT *"THIS STRING WONT BE LOADED"
```

Note that file declarations may not be made common between programs. The reasoning behind this restriction is that mis-alignment in file declarations could easily cause catastrophic destruction of the file structure under DOS. Therefore, whenever a program is loaded, all logical files are initialized to being closed and must be opened before any file I/O can occur. When chaining between programs, one should always close all files in which new space could have been allocated and then re-open the files in the next program.

## 4. PROGRAM CONTROL INSTRUCTIONS

DATASHARE normally executes statements in a sequential fashion. The program control instructions allow this flow to be altered depending on the state of the condition flags. There are five condition flags in DATASHARE: OVER, LESS, EQUAL, ZERO, and EOS. EQUAL and ZERO are two names for the same flag. Only the numeric and character string manipulating instructions, the READ instruction, and the READKS instruction alter the states of these flags. Reference should be made to the individual instruction explanations for the meanings of the flags.

## 4.1 GOTO

The GOTO instruction transfers control to the program statement indicated by the label following the instruction:

```
GOTO CALC
```

causes control to be transferred to the instruction labeled CALC.

The GOTO instruction may be made conditional by following the label by the preposition IF and one of the condition flag names. For example:

```
GOTO CALC IF OVER
```

will transfer control to the instruction labeled CALC if an overflow occurred in the last arithmetic operation. Otherwise, the instruction following the GOTO is executed.

The sense of the condition can be reversed by inserting the word NOT before the condition flag name as follows:

```
GOTO CALC IF NOT OVER
```

meaning control is transferred only if the overflow did not occur.

## 4.2 BRANCH

The BRANCH instruction transfers control to a statement specified by an index. For example:

```
BRANCH N OF START,CALC,POINT
```

causes control to be transferred to the label in the label list pointed to by the index N (i.e. START if N = 1, CALC if N = 2, and POINT if N = 3). If N is negative, zero, or larger than the number of labels in the list, control continues with the following statement. The index is



truncated to no decimal places before it is used (1.7 = 1).

The BRANCH instruction statement may be continued to the next line by the use of a colon in place of one of the variable delimiting commas. For example:

```
BRANCH N OF LOOP, START, READ1, WRITE1:
      WEOF1,STOP
```

#### 4.3 CALL

The CALL instruction is very similar to the GOTO instruction except that when a RETURN instruction is encountered after a transfer, control is restored to the next instruction following the CALL instruction. CALL instructions may be nested up to 8 deep. That is, up to eight CALL instructions may be executed before a RETURN instruction is executed. Being able to call subroutines eliminates the need to repeat frequently used groups of statements. Note, however, that in DATASHARE the space allowed for a program is very large and that, due to the virtual nature of this space, calling a subroutine is considerably more time consuming than executing the code in line if a page swap is invoked by the subroutine call. Therefore, in many cases it is much better to put some code in line instead of making it a subroutine, especially if the amount of code is quite small (say, less than a dozen lines). This is a trade-off which should be considered when one is dealing with code that will be executed very often (for instance, code that is executed every time a data item is entered). CALL instructions may be made conditional like the GOTO instruction. For example:

```
CALL FORMAT
CALL XCOMP IF LESS
```

#### 4.4 RETURN

The RETURN instruction is used to transfer control to the location indicated by the top address on the subroutine call stack. This instruction has no operand field but may be made conditional. For example:

```
RETURN
RETURN IF ZERO
```

#### 4.5 STOP

The STOP instruction causes the program to terminate and return to the MASTER program for that port. This instruction has no operand field but may be made conditional. For example:

```
STOP
```

## STOP IF NOT EQUAL

## 4.6 CHAIN

The CHAIN instruction causes the program, whose DOS name (with extension DS3) is in the literal or specified string, to be loaded and for control to be passed to its first executable statement. The characters used for the name start from under the formpointer of the specified string variable (or with the first quoted character in the case of a literal) and continue until either the logical end of the string has been reached or eight characters have been obtained. If the end of the string is reached before eight characters are obtained, the rest of the characters are assumed to be spaces. All DATASHARE 3 program object files are of extension DS3. The character after the 8th in the name variable (or the character after the logical length if the name is less than 8 characters long) is used as the drive number specification for the file. If the character is not an ASCII 0, 1, 2, or 3 or no character physically exists past the name, no drive specification is assumed and all drives starting with drive zero are searched when looking for the program name in the DOS directory (or directories). Otherwise, only the specified drive is searched for the name. For example, if in the following example NXXTPGM's formpointer was 4 and logical length was 6, the CHAIN command would try to load the program named "ROL/DS3" from drive 1.

```
NXXTPGM INIT "PAYROL1"
      .
      .
      CHAIN NXXTPGM
```

In the following example, however, the CHAIN command would try to load the program named "PAYROL1/DS3" off of any drive starting from the zeroth.

```
CHAIN "PAYROL1"
```

To make the CHAIN command try to load the program named "PAYROL/DS3" from drive one, one would have to execute the statement:

```
CHAIN "PAYROL 1"
```

since the 1 would appear after the eighth character in this case.

## 4.7 TRAP

TRAP is a unique instruction because, rather than taking action at the time it is executed, it specifies the

location to which a transfer of control (via the CALL mechanism) should occur if a specified event occurs during later execution. For example:

```
TRAP EMSG IF' PARITY
```

specifies that control should be transferred to EMSG if a parity failure is encountered during a READ or WRITE instruction. The control transfer is performed in a manner similar to the CALL instruction. Therefore, in the above example, if the parity error occurred during a disk READ instruction, the effect would be to insert a CALL EMSG instruction between the READ and the instruction immediately following it.

If an event occurs and the trap corresponding to that event has not been set, the message:

```
* ERROR * LLLLL X *      or
* ERROR * LLLLL X * Q
```

appears on the line currently positioned to on the terminal whose program caused the event. The first form appears for all traps except I/O traps. In the event of an I/O trap, a qualification letter is given where a "Q" is shown in the example (explained below under the "IO" trap). The LLLLL is the current value of the program counter and the X is an error letter. In most cases, LLLLL points to the instruction following the one that caused the problem. However, in certain I/O errors, LLLLL will point after the list item where the problem occurred. The following error letters can appear:

```
P - parity failure
R - record number out of range
F - record format error
C - chain failure
I - I/O error
B - illegal operation code
U - call stack underflow
A - interruptions already prevented
```

Note that the last three items shown above cannot be trapped. The B error will only show up if somehow an invalid object file is executed or if the system is failing. The U error will happen if the programmer forgets to perform a call or in some other fashion manages to execute a RETURN instruction without a corresponding CALL having been previously executed. The A error will happen if a PI instruction is executed while interrupts are currently prevented.

## PROGRAM CONTROL INSTRUCTIONS

The events that may be trapped are shown below. The capitalized name is the one used in the TRAP statement.

- PARITY - disk CRC error during READ or disk CRC error during write verification (the DOS retries an operation up to 5 times to get a good CRC before giving up and causing this event).
- RANGE - record number out of range (an access was made that was off the physical end of the file, a record was read which was never written, or a WRITAB was used on record which was never written)
- FORMAT - non-numeric data read into number (the read stops at the list item in error so the rest of the list items will not be changed)
- CFAIL - the specified program was not in the DOS directory or a ROLLOUT was attempted with one of the necessary system files missing
- IO - there is only one trap for all of the following conditions. Usually, however, the trap is used only for detecting whether a file exists or not. It is a good idea keep this trap clear whenever it is not being used specifically to detect the presence of a file to prevent confusion if one of the other conditions occurs. If the trap is not set then one of the following qualification letters indicates the nature of the I/O problem:
  - A - an access sequentially by key was attempted before any indexed sequential access was made using the logical file.
  - B - the READ mechanism ran off the end of a sector without encountering a physical end of record character (003).
  - C - an operation on a closed logical file was attempted.
  - D - a WRITE or INSERT indexed sequential operation was attempted where the specified key already exists in the index.
  - E - an EOF mark without at least four zero's was encountered.
  - I - the index file specified in an OPEN statement does not exist on the specified drive(s).
  - J - the index file found by the OPEN statement does not reside in the correct physical location on the disk (index files may never be moved, they must

- always be re-created).
- K - a null key was supplied in an operation where the key may not be null.
  - M - the data file specified in the OPEN statement does not exist on the specified drive(s).
  - N - the data file name specified in the OPEN or PREPARE statement was null.
  - O - the index file name specified in the OPEN statement was null.
  - P - the file specified in the PREPARE statement had some type of DOS protection (either write, delete, or both).
  - T - the tab value in the READ or WRITAB statement was off the end of the sector.
  - U - an EOF mark was encountered while a record was being deleted in the indexed sequential file.
  - V - one of the indexed sequential access overlays (DS/OV1, DS/OV2, or DS/OV3) could not be loaded by the DOS loader.
  - W - an index file pointer sector could not be read.
  - X - an index file header sector could not be read.
  - Y - the R.I.B. of the data file pointed to by the index file could not be read. (VWXY errors can be caused by parity errors, the drive being switched off line, or the disk cartridge being swapped with another while an operation is taking place.)

Note that the trap locations are cleared whenever a CHAIN occurs. Therefore, each program must initialize all of the traps it wishes to use. Also, whenever a certain event is trapped, the trap location for that event is cleared, which implies that, if the event is to be trapped again, its location must be reset by the trap routine.

#### 4.8 TRAPCLR

This instruction will clear the specified trap. For example:

```
TRAPCLR PARITY
```

will clear the parity trap previously set.

#### 4.9 ROLLOUT

The ROLLOUT feature allows execution of all ports currently on the DATASHARE system to be temporarily suspended while certain functions are performed under DOS.

When a ROLLOUT occurs, the program ROLLOUT/SYS will be run which writes system status and memory in a file called ROLLFILE/SYS. A beep is sounded at the console to alert the operator when a ROLLOUT is initiated. Clicks are sounded as ROLLFILE/SYS is created and another beep occurs when the file creation is completed. The DOS is then brought up at the console by the loading of programs SYSTEM0/SYS and INTRHAND/SYS. The ROLLOUT/SYS program then supplies the characters in the string specified by the Databus ROLLOUT instruction as if they were keyed in from the console (this will usually call the CHAIN program). When the DOS functions are completed, the DOS file DSBACK/CMD may be executed to restore the DATASHARE system to its previous status (this is usually the last program specified in the CHAIN file). DSBACK/CMD re-initializes the screen and then loads the ROLLFILE/SYS object file. This returns all ports to their previous point of execution when the ROLLOUT occurred.

An alternate to DSBACK/CMD is DSBACKTD/CMD which first requests the system operator to enter the time and date and then loads and executes the ROLLFILE/SYS object file. Note that since the purpose of DSBACKTD is to obtain the current time and date from the system operator, it is not logical to execute DSBACKTD from a CHAIN file. For this reason, DSBACKTD has been implemented such that it cannot be executed from a CHAIN file and also such that it will pause indefinitely until a response from the system console is obtained.

ROLLOUT/SYS, ROLLFILE/SYS, and INTRHAND/SYS are all provided on the DATASHARE interpreter system generation tape. A CFAIL trap will occur if ROLLOUT/SYS does not exist on disk, if ROLLFILE/SYS does not exist or is not big enough (must be at least 61 sectors), or if INTRHAND/SYS and SYSTEM0/SYS do not exist.

ROLLOUT is initiated by a DATASHARE program with the following instruction,

```
ROLLOUT (string variable)      or
ROLLOUT (string literal)
```

The string variable or literal specifies what function is initially to be executed under DOS and should be a command line acceptable to the DOS command handler. The string used is that in the variable from under the formpointer up to before a character that has a value less than 040 (octal), is a vertical bar (0174 octal), or has its sign bit set. In the normal case, this means the string used will be that from under the formpointer up through the physical length of the string. If it is desired for less than through the physical end of the string to be used, one should store a

vertical bar in the position after the last character to be used in the DOS command line string. A CFAIL trap will occur if the string variable is null. For example, the string's contents could be

#### CHAIN DSCFILE

When DOS is brought up by the ROLLOUT, the first thing to occur would be a chain to DSCFILE. The commands found in DSCFILE would then be executed (see user's guide on the DOS CHAIN command). DSCFILE could consist of these commands:

```
SORT AFILE,BFILE
SORT CFILE,DFILE
DSBACK
```

By using the CHAIN command, several DOS functions can be performed and the system automatically restored with the DSBACK command. If DSBACK is not included in the chain file, if the CHAIN aborted for some reason, if DOS was booted during the CHAIN, or if the string specified in the ROLLOUT consisted of a DOS function other than CHAIN, the DATASHARE system will have to be restored by the operator keying in DSBACK or DSBACKTD at the console.

The ROLLOUT feature is particularly useful when a file needs to be sorted with the DOS SORT command or an indexed file needs to be re-indexed using the DOS INDEX command. However, ROLLOUT may be very inconvenient to the users at other ports since execution of their programs will be suspended for at least 40 seconds. Note that the users at the other ports, unless informed of the fact, will not know what is happening when a ROLLOUT occurs. Since their terminals appear inactive, they may think the system has gone down for some other reason. Thus, consideration of other system users should be kept in mind when a ROLLOUT is used. Also, note that the time clock will be put behind however long the DATASHARE system is not executing unless it is updated through the use of the DSBACKTD command.

There are a number of precautions which must be observed during the use of ROLLOUT. The functions performed while under the DOS must not effect any of the operations that were taking place under the DATASHARE system. 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. The reason behind this is that when the DATASHARE operation is restored, certain items in memory reflecting the state of the DOS file structure will also be restored. If these items are no longer accurate in their reflection due to the fact that the file structure has been changed, terrible things can happen to the DATASHARE 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 command is executed (rolling the system back in will not see the configuration change). Typically, only one program will be active when a ROLLOUT is invoked, making it easy to observe all of these precautions.

#### 4.10 PI

This instruction (Prevent Interruptions) enables the programmer to prevent his background program from being interrupted for up to 20 Databus instruction executions. This instruction has no effect upon the foreground one millisecond interrupt which performs all port and printer I/O.

Normally, background execution can be interrupted by the INT key on the port terminal, by a higher priority execution being requested on another port (due to a foreground I/O process terminating), or by the time limit for the executing port running out. By executing the PI instruction, the programmer can postpone any of these interruptions for a specified number of instructions (up to a maximum of 20). This is particularly useful if one is trying to update a record in a file which could be updated at the same time by another port's program. By preventing interruptions from the time the record is read through when it is written, the programmer can be assured that the other program will not get in and modify the record in the mean time.

The number of instructions specified in the PI instruction is always a fixed decimal number (it may not be a numeric variable). For example:

```

PI          4
READ       F,KEY;PN,QTYONH,LOD
SUB        QTY FROM QTYONH
GOTO      NOTNUFF IF LESS
UPDATE    F;PN,QTYONH,LOD

```

Interruptions will be prevented from the PI instruction through the UPDATE instruction. Note that the number supplied to the PI instruction denotes the number of instructions after the PI instruction.

If a DISPLAY, KEYIN, CONSOLE, or PRINT instruction is executed while interruptions are prevented, the effect of the PI instruction is canceled (since execution of these instructions causes background execution for the given port



to cease). If a PI instruction is executed while interruptions are currently prevented, execution of the program is aborted with an error 'A' message. This prevents a program from being able to prevent interruptions for more than 20 instruction executions.

Note that when devising systems with complex data file structures one must always be prepared for his program being interrupted at any point in its execution without harming the file structuring beyond repair. Under DATASHARE 2, the fact that the operator could interrupt the program at any point by striking the INT key on the terminal necessitated procedures in the program which assured that the file structure could be recovered from such an interruption. One can now use the PI instruction to prevent the operator from causing such a disturbance. However, this should not be used by the programmer as a panacea for the interruption problem since interruptions can still be caused by power failures or the system operator restarting the processor. The PI can be very useful in preventing the operator from causing a situation which could require extensive recovery effort but the precautions which allow recovery in the event of an interruption at any point in the program must still be built in to allow recovery in the other less likely but still possible interruption cases.

#### 4.11 TABPAGE

This instruction allows the programmer to improve the execution speed of his program by letting him force sections of his program into certain pages of object code. Execution speed can be enhanced in this way because of the way the virtual storage mechanism for the object code works. The instruction consists only of the verb TABPAGE and has no operands (a label may be placed on a TABPAGE instruction line, however). See Section 12.1 for a more detailed discussion of the use of the TABPAGE instruction. Execution of the TABPAGE instruction causes control to be transferred to the first byte of the next page.

## 5. CHARACTER STRING HANDLING INSTRUCTIONS

Each string instruction, except LOAD and STORE, requires either one or two character string variable names following the instruction. (Note that the MOVE instruction is capable of moving strings to numbers, numbers to strings, and numbers to numbers, as well as moving strings to strings. See the following section and Section 6.5 for the entire description of the MOVE instruction. Also note that APPEND can move numbers into strings as well as strings into strings.) In the following sections, the first variable will be referred to as the source string and the second variable will be referred to as the destination string. In some cases, the source may be a literal. When it is, the formpointer always points to the first physical character in the string and the logical length always points to the last physical character in the string.

## 5.1 MOVE

MOVE transfers the contents of the source string into the destination string. Transfer from the source string starts with the character under the formpointer and continues through the logical length of the source string. Transfer into the destination string starts at the first physical character and when transfer is complete, the formpointer is set to one and the logical length points to the last character moved. The EOS flag is set if the ETX in the destination string would have been overstored and transfer stops with the character that would have overstored the ETX.

The MOVE instruction can also move character strings to numeric strings and vice versa. (The movement of numeric strings to numeric strings is covered in section 6.5.) A character string will be moved to a numeric string only if the character string from the formpointer through the logical length is of valid numeric format (only digits, spaces, a leading minus sign, and one decimal point allowed). Otherwise, the numeric string is not changed. Note that only the part of the character string starting with the formpointer is considered in the validity check and transferred if the string is of valid numeric format. The number in the character string will be reformatted to conform to the format of the numeric string. Rounding occurs if the number in the character string is too large to fit into the format of the numeric string (see Section 6 for rounding rules followed). The TYPE instruction (see section 5.14) is available to allow checking the character string for valid numeric format before using the MOVE instruction.

When a numeric string is moved to a character string, all characters of the numeric item (unless the ETX in the

CHARACTER STRING HANDLING INSTRUCTIONS

destination string would be overstored) are transferred starting with the physically first character in the destination string. When the operation is completed, the logical length is set to point to the last character transferred. The EOS condition is left true if the ETX of the destination string would have been overstored. In this case, transfer stops with the character before the one that would have overstored the ETX and the logical length is left pointing to the physical end of the string (which contains the last character transferred).

In the following examples, the logical length, formpointer, and content of each variable is shown before the statement is executed, the statement is shown and the contents of the variable that is changed by the execution of that statement is shown. The ^ denotes a space in the contents of a variable.

<u>VAR</u>	<u>LL</u>	<u>FP</u>	<u>Contents</u>	
STRING1	4	2	ABCDXLM	ETX
STRING2	6	3	DOGCAT	ETX
MOVE STRING1 TO STRING2				
STRING2	3	1	BCDCAT	ETX
STRING2	6	3	DOGCAT	ETX
MOVE "HELLO" TO STRING2				
STRING2	5	1	HELLOT	ETX
STRING1	9	3	AB100.327	ETX
NUMBER	0200		^39.00	ETX
MOVE STRING1 TO NUMBER				
NUMBER	0200		100.33	ETX

# CHARACTER STRING HANDLING INSTRUCTIONS

```

NUMBER    0200    100.33    ETX
STRING1   9   3    AB100.327  ETX
          MOVE NUMBER TO STRING1
STRING1   6   1    100.33327  ETX
    
```

Note that in the statement:

```
MOVE "ABC" TO NUMBER
```

the compiler will give an E error flag since it knows that this cannot be a valid operation (the move will not occur because the literal is not of valid numeric format). In the statement:

```
MOVE "2.3" TO STRING1
```

the compiler will generate a string to string move rather than a numeric to string move.

## 5.2 APPEND

APPEND appends the source string or number to the destination string. A numeric item is treated exactly as if it were a string with a formpointer pointing to the first physical character and a logical length pointing to the last physical character in the number. The characters appended are those from under the formpointer through under the logical length pointer of the source string. The characters are appended to the destination string starting after-the-formpointed-character in the destination string. The source string pointers remain unchanged, but the destination string pointers both point to the last character transferred. The EOS condition will be set if the new string will not fit physically into the destination string, but all characters that will fit will be transferred.

The following example shows two strings before the operation, the operation, and the result in the second string after the operation:

```

STRING1   8   6    JOHN^DOE    ETX
STRING2  11  11    MARY^JONES^^^^^^^^^^^^^^  ETX
          APPEND STRING1 TO STRING2
STRING2  14  14    MARY^JONES^DOE^^^^^^^^^^  ETX
    
```

## CHARACTER STRING HANDLING INSTRUCTIONS

The following example shows a destination string before the operation, an operation appending a literal to the destination string, and the destination string after the operation:

```

STRING2      8   9   MARY^JONES^^^^^^^^^^^^^^ ETX
      APPEND ".XX.YY." TO STRING2
STRING2     15  15   MARY^JON.XX.YY.^^^^^^ ETX
    
```

The following example shows the use of APPEND to move a numeric item into a string item:

```

NUMBER      0200   100.33   ETX
STRING      9   2   ABCDEFGHI ETX
      APPEND NUMBER TO STRING
STRING      9   8   AB100.33I ETX
    
```

### 5.3 MATCH

MATCH compares two character strings starting at the formpointer of each and stopping when the end of either operand's string is reached. If either formpointer is zero before the operation, the MATCH operation will result in only clearing the LESS and EQUAL flags and setting the EOS flag. Otherwise, the "length" of each string is calculated to be LENGTH-FORMPOINTER+1 and the LESS flag is set if the destination string length is less than that of the source string. The two strings are then compared on a character-for-character basis for the number of characters equal to the lesser of the two lengths. If all the characters match, the EQUAL flag is set. Otherwise, the LESS flag's meaning is changed to indicate whether the ASCII value of the destination character is less than the ASCII value of the source character (LESS flag set) or vice versa (LESS flag reset) for the first pair of characters that do not match. Some examples and their results follow:

<u>SOURCE</u>	<u>DESTINATION</u>	<u>RESULT</u>
ABCDE	ABCD	EQUAL, LESS
ABC	Z	NOT EQUAL, NOT LESS
ZZZ	AAA	NOT EQUAL, LESS
ABC	ABC	EQUAL, NOT LESS
ABCD	ABCDE	EQUAL, NOT LESS

Examples:

```
MATCH A TO B
MATCH STR1,STR2
```

#### 5.4 CMOVE

CMOVE moves a character from the source operand to under the formpointer in the destination string. The character from the source operand may be a quoted alphanumeric (note that the forcing character rule does not apply here), the character from under the formpointer of a string variable, or an octal control character (000 to 037). If either operand has a formpointer of zero, an EOS condition and no transfer occurs.

Examples:

```
CMOVE XDATA TO YDATA
CMOVE "A" TO CAT
CMOVE X,Y
CMOVE 015,Y
```

#### 5.5 CMATCH

CMATCH compares two characters, one taken from each of the source and destination operands. The characters may be quoted alphanumeric (note that the forcing character rule does not apply here), from under the formpointer of a string variable, or octal control characters (000 to 037). An EOS condition occurs if either formpointer is zero, and no other conditions are set. Otherwise, the EQUAL and LESS conditions are set appropriately. The LESS condition is set if the destination string character is less than the source string character.

Examples:

```
CMATCH XDATA TO YDATA
CMATCH "A",DOG
CMATCH CAT TO "B"
CMATCH 015,DOG
```

#### 5.6 BUMP

BUMP increments or decrements the formpointer of the first operand if the result will be within the string (between 1 and the logical length). If no parameter is supplied, BUMP increments the formpointer by one. However, a positive or negative literal value may be supplied to cause the formpointer to be moved in either direction by any amount. The EOS flag will be set and no change in the formpointer occurs if it would be less than one or greater

than the logical length after the movement had occurred.

Examples:

```
BUMP CAT
BUMP CAT BY 2
BUMP CAT,-1
```

## 5.7 RESET

RESET changes the value of the formpointer of the source string to the value indicated by the second operand. If no second operand is given, the formpointer will be reset to one. The second operand may be a quoted character, in which case the ASCII value minus 31 (space gives one, ! two, " three, etc) will be used for the value of the formpointer of the source string. The second operand may also be a character string, in which case the ASCII value minus 31 of the character under the formpointer of that string will be used for the value of the formpointer of the source string. The second operand may also be a numeric string, in which case the value of the number will be used for the formpointer of the source string.

The use of a string variable as the second operand in a RESET instruction may not be obvious at first. One application could be in doing code conversions where each character in the string to be converted is used as a formpointer value in a code conversion string from which is picked to corresponding converted character to be used as the character in the converted string. Another use is in the coding of item positions within a string into a single character. For example, in a file one might want to place an item in a variable location within the record. The first character of the record could be a character which corresponds to the column position within the record of the start of the item. One could read the first character of the record into a one character string variable and then the rest of the record into a large string variable. The large string variable could then have its formpointer reset to the position indicated by the first character in the record and the item could then be moved to another variable with the MOVE instruction.

RESET also has the capability of extending the logical length of the first operand. If the formpointer value specified is past the logical length of the first operand, the logical length will be extended until it will accommodate the formpointer value. If this would cause the logical length to be past the physical end of the string, the logical length and formpointer will both be left pointing to the last physical character in the string. This feature is useful in extracting and inserting information within a

large string. The EOS condition will be set if a change in the logical length of the first operand occurs.

Examples:

```

RESET XDATA TO 5
RESET Y
RESET Z TO NUMBER
RESET Z TO STRING
    
```

Note that the RESET instruction is very useful in code conversions and hashing of character string values as well as large string manipulation.

#### 5.8 ENDSET

ENDSET causes the operand's formpointer to point where its logical length points.

Example:

```

ENDSET PNAME
    
```

#### 5.9 LENSET

LENSSET causes the operand's logical length to point where its formpointer points.

Example:

```

LENSSET QNAME
    
```

#### 5.10 CLEAR

CLEAR causes the operand's logical length and formpointer to be zero. None of the data characters are changed.

Example:

```

CLEAR NBUFF
    
```



5.11 EXTEND

EXTEND increments the formpointer, stores a space in the position under the new formpointer, and sets the logical length to point where the new formpointer points if the new logical length would not point to the ETX at the end of the character string. Otherwise, the EOS flag is set and no other action is taken.

Example:

EXTEND BUFF

5.12 LOAD

LOAD performs a MOVE from the character string pointed to by the index numeric string, given as the second operand, to the first character string specified. The instruction has no effect if the index is negative, zero, or greater than the number of items in the list. Note that the index is truncated to no decimal places before it is used (e.g. 1.7 = 1).

Example:

LOAD AVAR FROM N OF NAME,TITLE,HEDING

5.13 STORE

STORE performs a MOVE from the first character string specified to a character string in a list specified by an index numeric variable given as the second operand. The instruction has no effect if the index is negative, zero, or greater than the number of items in the list. Note that the index is truncated to no decimal places before it is used (e.g. 1.7 = 1).

Examples:

STORE Y INTO NUM OF ITEM,ENTRY,ALINK  
STORE "XX" INTO NUM OF A1,A2,A3

The LOAD and STORE instructions may be continued to the next line by the use of a colon:

Examples:

LOAD SYMBOL FROM N OF VAR,CONST,DEC:  
COUNT,FLAG,LIST  
STORE NAME INTO NUM OF A,B,C,D,E,F,G:  
H,I,J,K,L,M

# CHARACTER STRING HANDLING INSTRUCTIONS

## 5.14 CLOCK

CLOCK enables the programmer to access the DATASHARE system time clock, day, and year information. This information is initialized by the operator when DATASHARE is activated and then kept current by a foreground program driven by the one millisecond interrupt clock. This interrupt is accurate to approximately 0.005 percent or four seconds per day. There are three variables that the CLOCK instruction can access. These are given the names TIME, DAY, and YEAR. All are character strings with TIME being in the format:

12:34:56

and ranging from 00:00:00 to 23:59:59, DAY being in the format:

123

and ranging from 001 to 365 (except to 366 on leap years), and YEAR being in the format:

12

and ranging from 00 to 99, being the last two digits of the year. Note that when the TIME goes from 23:59:59 to 00:00:00, the day is incremented. The new day value is not checked to be a valid Julian date, however, implying that the system must be manually reset at midnight at the end of the year. The CLOCK instruction performs a character string to character string move with the special variable in the source and the character string to receive the information in the destination operand specification. Note that the user's program may have variables called TIME, DAY, and YEAR.

For example:

```
CLOCK    TIME TO TIME
CLOCK    DAY TO DAY
CLOCK    YEAR TO YEAR
```

would move the information in the system variables into user defined variables called TIME, DAY, and YEAR also.

Note that the clock value is not allowed to be updated by the foreground interrupt during the actual transfer of characters from the system data into the user's data item. However, an interrupt could occur between the time one clock item was moved and the next, thereby necessitating a precaution if one is to obtain both the time and the day figure. For example, if the time was 23:59:59 and one moved

## CHARACTER STRING HANDLING INSTRUCTIONS

the TIME into a variable and then the foreground interrupted and caused the clock to be incremented to the next second, the TIME would then read 00:00:00 and the DAY would have been incremented. If one then obtained the DAY figure, he would have the wrong day for the time he had gotten. Therefore, when obtaining both the TIME and DAY, one must first get the DAY, then get the TIME, and then go back and make sure the DAY had not changed. For example:

```
CLOCK      DAY TO DAY
CLOCK      TIME TO TIME
CLOCK      DAY TO DAY2
COMPARE    DAY TO DAY2
GOTO       TIMEOK IF EQUAL
CLOCK      DAY TO DAY
TIMEOK     (etc)
```

The system brings itself up automatically one minute after it is started if an operator does not attend the system console. In this case, all CLOCK items are initialized to zero. Therefore, one can determine that the CLOCK items were not initialized by examining the DAY string and checking for a value of 000.

### 5.15 TYPE

TYPE sets the EQUAL condition if the string contained from the formpointer through the logical length of the specified string variable is of valid numeric format (only leading minus, one decimal point, and digits or spaces).

## 6. ARITHMETIC INSTRUCTIONS

All of the arithmetic instructions have certain characteristics in common. Except for LOAD and STORE, each arithmetic instruction is always followed by two numeric string variable names. The contents of the first variable is never modified and, except in the COMPARE instruction, the contents of the second variable is always the result of the operation. For example, in:

ADD XAMT TO YAMT

the content of XAMT is not changed, but YAMT contains the sum of XAMT and YAMT after the instruction is executed.

Following each arithmetic instruction, the condition flags OVER, LESS, and ZERO (or EQUAL) are set to indicate the results of the operation. OVER indicates that the result of an operation is too large to fit in the space allocated for the variable (a result is still given with truncation at the left and rounding at the right, however). LESS or ZERO (EQUAL) indicates respectively that the content of the second variable is negative or zero following the execution of the instruction (or would have been in the case of COMPARE).

Whenever overflow occurs, the higher valued digits that do not fit the variable are lost. For example, if a variable is defined:

NBR42    FORM 2.2

and a result of 4234.67 is generated for that variable, NBR42 will contain only 34.67.

Whenever an operation produces lower order digits than will fit in the destination variable, the result is rounded up if the digit to the right of the last one that would fit is greater than 4 (standard rounding rules). A variable with the FORM 3.1 would contain:

46.2	for	46.213
812.5	for	812.483
3.7	for	3.666
3.9	for	3.850
632.0	for	4632

with the OVER condition occurring for only the last result.

Note that if an OVER occurs during an ADD, SUB, or COMPARE of two strings of different physical lengths, the result will not and the LESS condition flag may not be correct.

## 6.1 ADD

ADD causes the content of variable one to be added to the content of variable two:

Examples:

```
ADD X TO Y
ADD DOG,CAT
ADD "1", LEN
```

## 6.2 SUB or SUBTRACT

The SUB instruction (the compiler will also accept a mnemonic of SUBTRACT) causes the content of variable one to be subtracted from the content of variable two.

Examples:

```
SUB RX350 FROM TOTAL
SUB "32.5" FROM RATE
SUBTRACT Z,TOTAL
```

## 6.3 MULT or MULTIPLY

The MULT instruction (the compiler will also accept a mnemonic of MULTIPLY) causes the content of variable two to be multiplied by the content of variable one. The restrictions mentioned in the introduction about the length of multiplication operands are that the sum of the number of characters in the two operands must be less than 32.

Examples:

```
MULT B BY A
MULT ".005" BY TOTAL
MULTIPLY W,Z
```

## 6.4 DIV or DIVIDE

The DIV instruction (the compiler will also accept a mnemonic of DIVIDE) causes the content of of the second variable to be divided by the content of the first variable. The restriction upon division operands is that the number of characters in the dividend plus the number of characters in the divisor plus two times the number of characters after the decimal point in the divisor must be less than 32. Division by zero results in the OVER condition being set and the destination variable not being changed.

If the quotient cannot be represented fully in the destination variable format, the quotient will be rounded to the number of places in the destination variable if the

divisor has at least one digit place after the decimal point. If there are no digit places after the decimal point in the divisor, the quotient will be truncated (rounded down) to the number of places in the destination variable.

Examples:

```
DIV SFACT INTO XRSLT
DIV "3.0" INTO QUANTITY
DIVIDE X3,HOURS
```

#### 6.5 MOVE

MOVE causes the content of variable one to replace the content of variable two.

Examples:

```
MOVE FIRST TO SECOND
MOVE "0" TO COUNTER
MOVE A,B
```

#### 6.6 COMPARE

COMPARE does not change the content of either variable but sets the condition flags exactly as if a SUB instruction has occurred.

Examples:

```
COMPARE XFRM TO YFRM
COMPARE "100" TO LINENR
COMPARE TIME1,TIME2
```

#### 6.7 LOAD

The LOAD instruction selects the numeric string variable out of a list based on a numeric index variable. It then performs a MOVE operation from the contents of the selected variable into the first operand. If the index is negative, zero, or greater than the number of items in the list, then the instruction has no effect. Note that the index is rounded to no decimal places before it is used (e.g. 0.1 = 0).

Example:

```
LOAD CAT FROM N OF CAT,MULT,SPACE
```

#### 6.8 STORE

The STORE instruction selects a numeric string variable from a list based on the value of a numeric index variable. It then performs a MOVE operation from the contents of the

first operand into the selected variable. If the index is negative, zero, or greater than the number of items in the list, the instruction has no effect. Note that the index is rounded to no decimal places before it is used (e.g. 0.1 = 0).

Example:

```
STORE X INTO NUM OF VAL,SUB,TOT
```

The LOAD and STORE instruction statements may be continued to the next line by the use of a colon.

Examples:

```
LABEL   LOAD NUMBER FROM N OF N1,N2,N3,N4,N5:  
        N6,N6,N8,N9  
ENTRY   STORE COUNT INTO NUM OF T1,RATE,DIST:  
        SPD,COST,TOT,SUM  
        STORE "2.3" INTO X OF N1,N2,N3
```

## 7. INPUT/OUTPUT INSTRUCTIONS

The DATASHARE statements that move data between the program variables and the terminal, printer, or disk, allow a list of variables to follow the operation mnemonic. This list may be continued on more than one line with the use of a colon. Continuation is encouraged over repeating the operation on sequential lines because of the resulting increase in execution speed. The reason for this is that DATASHARE performs all terminal and printer I/O with interrupt driven routines which execute the entire I/O statement before having to return control to the background program. The interrupt driven routine executes entirely out of main memory while the background usually involves some page swapping due to the virtual nature of its program storage. If several I/O statements are given sequentially, the background program will have to be swapped in for each statement. However, if the entire operation has been performed with one I/O statement, background swapping would not have occurred until the operation was complete. This greatly increases execution speed.

The I/O list may contain some special control information besides the names of the variables to be dealt with. It may also include octal control characters (000 through 037). DATASHARE has no formatting information in its input and output operations other than the list controls and that implied by the format of the variables. The number of characters transferred is always equal to the number of characters physically allocated for the string (except in some special cases) allowing the programmer to set up his formatting by the way he dimensions his data variables.

### 7.1 KEYIN

KEYIN causes data to be entered into either character or numeric strings from the keyboard. A single KEYIN instruction can contain many variable names and list control items. When characters are being accepted from the keyboard, the flashing cursor is on. At all other times the cursor is off.

When a numeric variable is encountered in a KEYIN statement, only an item of a format acceptable to the variable (not too many digits to the left or right of the decimal point and no more than one sign or decimal point) is accepted. If a character is struck that is not acceptable to the format of the numeric variable, the character is ignored and a bell character is returned (causing a beep on a Datapoint CRT terminal). Note that if fewer than the allowable number of digits to the left or right of the decimal point are entered, the number entered will be reformatted to match the format of the variable being



entered. When the ENTER key is struck, the next item in the instruction list is processed.

When a character string variable is encountered, the system accepts any set of ASCII characters up to the limit of the physical length of the string. The formpointer of the string variable is set to one and characters are stored consecutively starting at the physical beginning of the string. When the ENTER key is struck, the logical length is set to the last character entered and the next item in the keyin list is processed. If the ENTER key is struck without any other characters having been entered (a null string is entered), both the logical length and form pointer of the string are set to zero. The program can check for a variable with a null entry by checking for an EOS condition after doing a RESET or CMATCH instruction on the variable in question (see Section 5.5 or 5.7).

Other than variable names, the KEYIN instruction may contain quoted items, list controls, and octal control characters (000 to 037). Quoted items are simply displayed as they are shown in the statement. The list controls begin with an asterisk and allow such functions as cursor positioning and screen erasure. The \*P<n>:<m> control positions the cursor to horizontal position <n> and vertical position <m>. Note that these numbers may either be literals or numeric variables and both positions must always be given in a \*P command. The horizontal position is restricted by the interpreter to be from 1 to 80 and the vertical position is restricted to be from 1 to 24. Numbers outside this range have the effective value of 1. (If lines 13 through 24 are positioned to when the console is being used for the port, the positioning will be to line 12.)

The \*ES control positions the cursor to 1:1 and erases the entire screen, the \*EF control erases the screen from the current cursor position, the \*EL control erases the rest of the line from the current cursor position, the \*C control causes the cursor to be set to the beginning of the current line, the \*L control causes the cursor to be set to the following line in the current horizontal position, the \*N control causes the cursor to be set to the first column of the next line, and the \*R control causes the screen to roll up if port one is using the system console for its terminal.

The control characters in the KEYIN instruction are output according to their ASCII meaning. They are only useful for teletype, UNITERM, and message switching applications and should not be sent to the Datapoint 3360-102.

Normally, the cursor is positioned to the start of the next line at the termination of a KEYIN statement. However,

placement of a semicolon after the last item in the list will cause this positioning to be suppressed, allowing the line to be continued with the next KEYIN or DISPLAY statement. This feature is also true of the PRINT command.

Example:

```
KEYIN *ES,"NAME: ",NAME,*P35:1,"ACNT NR: ":
      ACTNR," ADDRESS: ",STREET,*P10:3:
      CITY,*PX:4,"ZIP: ",ZIP;
KEYIN "ABC",021,NVAR
```

During a KEYIN, any unrecognizable characters (not in the printing ASCII set) sent in from the terminal will be ignored and a beep returned. Also, a mode called keyin continuous is available (turned on with list control \*+ and turned off with list control \*- or the end of the statement) which causes the system to react as if an ENTER key had been struck when the operator enters the last character that will fit into a variable. This mode allows the system to react in much the same way as a keypunch machine with a control card.

While keying a given variable, the operator can strike the BACKSPACE key (control H on Teletype) and cause the last character entered to be deleted. He may also strike the CANCEL key (control X on Teletype) and cause all of the characters entered for that variable to be deleted.

A circular input buffer allows the operator to send up to seven characters from the keyboard before they are requested by the system. Note that there is no feedback at this level as the characters are fed back only as they are taken from the buffer. This buffer allows the operator to continuously enter data without having minor delays in the response of the system break his stride.

A special case of KEYIN is the interrupt character, the INT key on a Datapoint 3360-102 (control shift L on a Teletype machine and CANCEL with both the KEYBOARD and DISPLAY keys depressed on the system console). Normally, when the cursor is not flashing, all characters will be ignored (not accepted from the seven character circular input buffer) until input is requested. The exception, however, is the interrupt character, which may be keyed at any time (it will be postponed if a PI instruction is in effect) and will result in an immediate CHAIN to the MASTER program. Thus, the currently executing program will stop, the printer (if being used by the terminal) will be RELEASED, and the MASTER program will begin execution.

Another special case of KEYIN is the NEW LINE character which is the NEW LINE key on the Datapoint 3360 (shift O on

the Teletype and the DEL or underline character on the system console). If this key is struck during a KEYIN statement, the current variable is terminated as if the ENTER key was struck and all subsequent variables in the statement will be set to zero or their form pointers and logical lengths set to zero depending on whether they are numeric or string variables. Control will fall through to the next DATASHARE statement.

The list control, \*T, may be included in the KEYIN statement causing a time out if more than two seconds elapse between the entry of two characters. The time out has the same results as if the NEW LINE key had been struck. This function is useful for message switching applications.

The list control, \*W, may be included in the KEYIN statement causing a one second pause at that point in the list sequence. This control is especially useful in programs which wish to simply pause for a number of seconds. Any number of seconds of pause may be achieved by simply putting in the required number of \*W controls in the list. A pause using \*W imposes very little overhead upon the system.

The list controls \*EOFF and \*EON may be included in the KEYIN statement causing the echo of entered characters to be inhibited or enabled respectively. When echo is inhibited, the KEYIN statement causes only the characters specifically mentioned in the list to be transmitted back to the terminal. Therefore, the statement:

```
KEYIN *EOFF,INLINE;
```

would allow the variable INLINE to be entered from the terminal with absolutely no characters being transmitted to the terminal. Since the cursor on and off controls will not be transmitted to the terminal, there will be no indication in this case that input is being requested. The echo inhibit is useful for message switching applications but could also be used where passwords are to be entered and it is desired to suppress their display. In this case, the statement:

```
KEYIN *EOFF,*P1:10,"ENTER PASSWORD: ":  
022,PASSWORD,024
```

could be used. Note that even though echo is inhibited, the cursor positioning and literal characters are still transmitted to the terminal since they are specifically mentioned. Notice also that the carriage return and line feed will be sent at the end of the statement since a semi-colon is not supplied. The 022 character is a cursor on for the 3360-102 and the 024 is a cursor off (this also

works for the system console since when port one is being run on the system console a program is being executed which actually interprets all of the 3360-102 control codes to perform the appropriate action on the console screen). The cursor controls must be specifically mentioned since the echo inhibit prevents them from being sent automatically. The echo is always enabled at the conclusion of the KEYIN statement. Therefore, one must always inhibit the echo at the start of each statement in which no echo is desired.

## 7.2 DISPLAY

DISPLAY follows the same procedure as KEYIN except that when a variable name is encountered in the list following the instruction, the variable's contents is sent to instead of being requested from the terminal. Character strings are displayed starting with the first physical character and continuing through the logical length. Spaces will be displayed for any character positions that exist between the logical length and physical end of the string unless the \*+ mode (keyin continuous in the KEYIN instruction) is active, in which case no more characters are put out after the logical length. Numeric strings are always displayed in total. Quoted strings, list controls, and octal control characters may be included in the display instruction and are handled in the same manner as described for the KEYIN instruction. Note that the \*T, \*EON, and \*EOFF controls will simply be ignored in the DISPLAY statement.

Examples:

```
DISPLAY *P5:1,"RATE: ",RATE:
          *P5:2,"AMOUNT: ",AMNT
DISPLAY "ABC",021,S1;
```

## 7.3 CONSOLE

CONSOLE is similar to DISPLAY except the output is on the system console (2200 display screen) instead of the terminal. (Note that all output to the system console is inhibited if it is being used as the terminal for port one. In this case, all statements that would cause display on the console execute as if the display were occurring but no display actually occurs.) The output always is on the line assigned for the terminal executing the CONSOLE instruction and therefore any vertical positioning of the cursor is ignored. The only DISPLAY list control that are effective are \*P (cursor positioning) and \*EL (erase to end of line). All others are simply ignored. A CONSOLE statement which begins without positioning will start displaying at column five on the appropriate port line at the console. If positioning is specified, \*Px:y, y is ignored and x may be any number from 1 through 80. Thus, the port number and

asterisk appearing in column 1 through 4 on the CONSOLE may be overwritten. If the horizontal position is out of the allowed range, position 1 is assumed. If the display flows over the 80 character limit, the extra characters will not be displayed. If the CONSOLE statement is not terminated by a semi-colon, the carriage return and line feed is ignored but two spaces are put out after the last character displayed. The CONSOLE instruction is useful in alerting the system operator (if such a person exists) to some condition in the program. The 2200 screen also displays at the left the state of the carrier detection signal (an asterisk appears in column 1 if the carrier is being detected) from each terminal and the name of the program to which a CHAIN was last executed. The current wall clock time is displayed in the upper right corner of the screen.

Example:

```
CONSOLE *P20:1,"OPERATOR ALERT"
```

#### 7.4 BEEP

BEEP causes an ASCII "ring bell" character to be sent to the terminal.

Example:

```
BEEP
```

#### 7.5 PRINT

DATASHARE supports either one local printer or one servo printer (depending upon how the system is configured). The printer may be accessed on a sequentially shared basis by any of the eight terminals. If the printer is being used by another terminal when the given terminal executes a PRINT statement, the given terminal will be suspended until the printer becomes available, or until the interrupt character is keyed.

The PRINT instruction causes the contents of variables in the list to be printed in a fashion similar to the way DISPLAY causes the contents of variables to be displayed. The list controls are much the same as DISPLAY except that cursor positioning cannot be used, column tabulation is provided (\*<n> causes tabulation to column <n> unless that column has been passed) and \*F causes an advance to the top of the next form. Octal control characters may also be included in the print instruction. The PRINT statement may be continued on more than one line by the use of a colon.

## Examples:

```

PRINT DATE,*20,"TRANSACTION SUMMARY",*C,*L:
      PNAME,*N,*10,RATE,*20,HOURS,*30:
      AMT,*L
PRINT "ABC",021,S1;

```

If the servo printer is configured into the system, the available number of bytes in the user data area is 512 less. In addition, the tabbing in the PRINT statement can move the carriage in the reverse direction and any sequence of printer controls will be executed in precisely the sequence specified. For example, one could print 10 characters, tab back to column 5 and overprint that column, do one line feed, and print five characters which would appear in columns 6 through 10 under the first line. He could then do a form feed and print 10 more characters which would appear in columns 11 through 20 at the top of the next page. One must be careful not to do these things, however, if he plans to use the same program with non-servo printers.

If the servo printer is being used, the paper out condition will be checked whenever a top of form control is given in a PRINT statement. If, after the top of form function is performed, the paper out condition is present, the console will make a uniquely characteristic beeping sound to alert the system operator that more paper must be placed in the printer. The beeping sound will stop when the front cover of the printer is swung out but will resume if the cover is replaced to its original position with the paper out indicator still on. The recommended procedure is to open the front cover, remove the last form still in the printer, place new paper in the printer with the top of the form aligned with the print head, and finally close the front cover.

Another feature allowed with the servo printer is minor vertical spacing (there are eight minor vertical spaces for one standard line space). Control characters either given directly in the PRINT statement or contained within a string variable can cause the paper to be fed either up or down up to seven minor vertical spaces. The characters zero through seven cause the paper to be fed down the page (the normal spacing direction) a corresponding number of minor spaces. The characters eight through fifteen cause the paper to be fed up the page (opposite to the normal spacing direction) zero through seven minor spaces respectively. The characters sixteen through twenty-two cause the carriage to move to the left seven through one column positions respectively (horizontal minor positioning cannot be performed). The character twenty-three causes no printer action. The characters twenty-four through thirty one cause

the carriage to move to the right one through eight column positions respectively. This feature on the servo printer allows different kinds of underscoring and super- and/or sub-scripting in the printed output.

#### 7.6 RELEASE

The RELEASE instruction ends a user's control of the printer and causes the printer to advance to the top of the next form. When RELEASE is executed by a user, another user that has been waiting for the printer will gain its control. When a user disconnects from the system or keys the INT character on his terminal (interrupt key), the printer is automatically released.

## 7.7 Disk I/O

DATASHARE 3 allows a large variety of file structures and access methods. The structures can be dependent upon the physical sectoring of the disk, physically sequential, or logically indexed. The access methods can be physically random, physically sequential, logically random, or logically sequential with any mix of these being allowed on logically indexed files. This section will describe the various file structures that can be created, how positioning is maintained within these structures, and how access to desired information within the structure can be achieved. It will then describe the various operations that can be performed upon the information within the file.

### 7.7.1 File structures

The most basic structure within a file is a physical record. A physical record can contain at most 249 data characters (note that there is no decimal number compression within any of these file structures so a number always occupies the number of characters that are contained within the FORM which defines the number). A physical record corresponds to exactly one physical sector on the disk and is always terminated by a 003 character.

The next level of structuring is a logical record. Depending upon the way the user structures his file there may or may not be an integral number of logical records within a physical record. A logical record is terminated by a 015 character after which another logical record begins. Note that logical records can extend across physical record boundaries (terminated by 003 characters) so that a file with logical records may appear in the first two physical records as follows (the items in parenthesis are the logical and physical record termination characters):

```
01128558382 AASDFQWERKFKDSKA (015) 1234848 (003)
8483 LAKSJDFLKASDFKKJ (015) 48828388483 KI (003)
```

Note that the first logical record extends about two thirds of the way through the first physical record and is then terminated by the 015 character. The first seven characters of the second logical record are also contained in the first physical record at which point the first physical record is terminated. The rest of the second logical record extends about half way through the second physical record and is then terminated by the 015 character. At this point the third logical record starts and so on.

Also note that there is no restriction upon the length of a logical record (a single logical record may extend across many physical records) but that it is a good idea to



keep logical records reasonably short to prevent them from becoming hard to deal with. If one had wanted to keep only one logical record per physical record he would have made the file appear as follows:

```
01128558382 AASDFQWERKFKDSKA (015) (003)
12348488483 LAKSJDFLKASDFKKJ (015) (003)
48828388483 KILKJLKJLKSJDFKD (015) (003)
```

Note that it took more disk space to store the same amount of information in this case than in the previous case. It is sometimes desirable to give up this space in return for the capability of using the fastest accessing method of directly accessing physical records (access methods are discussed in Section 7.7.2). A structure which allows logical records to cross physical record boundaries is called a record compressed structure.

In some data files large numbers of contiguous spaces appear. These files can be compressed even further than simple use of record compression by the use of space compression (the general purpose DOS editor, the DOS SORT program, a number of the terminal emulator programs, the DATASHARE compiler (listing file output), and DATASHARE programs can all generate space compressed records). A space compressed structure appears much like a record compressed structure except for the addition of the 011 control character. This control indicates that the next byte is a positive 8-bit binary word which tells how many spaces were replaced by the compression code character pair. This number will never be less than 2 (since it is wasteful to expand one or zero spaces into two characters) and may be as large as 255. In addition, the 011 will never appear as the last character in a physical record since the character indicating the number of spaces will always appear after the 011 (otherwise the 003 indicating the end of the physical record and three spaces compressed could not be differentiated). For example, in the following a logical record is shown first without space compression and then with space compression:

```
NOW IS THE TIME FOR (015)
NOW IS THE(011)(002)TIME(011)(007)FOR (015)
```

The second record is physically shorter than the first by six characters. It may seem silly to compress two spaces into a two character compression code but most programs do this because it is logically simpler to program. If more than 255 contiguous spaces appear in the data record, multiple space compression codes will appear. Space compressed records are most useful where large numbers of spaces appear in the file (as in print files) and where the records are not to be modified in place. If the record is

to be modified in place, space compression is discouraged since the number of spaces could change and the physical length of the logical record could change.

A file which can be accessed physically sequentially must not have any physical records without the proper format between the beginning of the file and an end of file mark. The end of file mark always starts at the beginning of a physical record and contains exactly six 000 characters followed by the physical record termination character (003). The rest of the characters in the physical record are of no significance. Note that if there are no physical records besides the one containing the end of file mark, the file would be null (which is a valid condition for a file).

A physically sequential data file can be logically indexed. One cannot tell that a file is indexed by looking only at the data file since the indexing information is maintained in a separate file called the index file (and usually of DOS extension ISI). The index file contains the name and extension of the data file which it indexes and a set of keys and pointers which relate the key value of a logical record to its physical position within the data file. DOS utilities exist for the creation of the index file which must always be performed outside of the DATASHARE interpreter.

The index file is a n-ary tree where n is determined by the length of the key and where there are enough levels to make the top node in the tree always fit within one disk sector (contain at most n branches). One can conservatively estimate the number of sectors that will be used in the index file by the following method. The actual number used may be less because trailing spaces in keys are discarded and more than the minimum number of keys may fit in a sector.

To compute the index file length, divide 250 by the key length plus 7 and discard the remainder (do not round up the result). This number should then be divided into the number of logical records to be indexed and the answer rounded up (if the remainder is non-zero then add one to the answer and discard the remainder). Save this number which is the number of sectors at the lowest level of the index tree. Then divide 250 by the key length plus 3 and discard the remainder. This number should then be divided into the number saved before the previous step and the answer rounded up. Save this number which is the number of sectors at the next higher level of the index tree. If the answer produced is greater than one, repeat the previous step (dividing 250 divided by the key length plus 3 into the previous answer). When the answer has been reduced to one, total all of the numbers of sectors required for each level and the result

will be the total number of sectors required in the index file.

For example, assume that the data file contains 10000 logical records and the key is 10 characters long. The first computation is  $250/(10+7) = 14.71$  or 14 discarding the remainder. The next computation is  $10000/14 = 714.29$  or 715 rounding up. Therefore, the lowest level of the index tree will require 715 sectors. The next computation is  $250/(10+3) = 19.23$  or 19 discarding the remainder. The next computation is  $715/19 = 37.63$  or 38 rounding up. Therefore, the next higher level of the index tree will require an additional 38 sectors. The next computation is  $38/19 = 2.00$  or 2 rounding up. Therefore, the next higher level of the index tree will require an additional 2 sectors. The next computation is  $2/19 = 0.11$  or 1 rounding up. Since one sector has been reached, the totals are made:  $715+38+2+1 = 756$  sectors for the entire index tree.

### 7.7.2 Positioning and accessing

In DATASHARE, all files are referenced by way of logical files. These files are declared in the data area of the program using the FILE and IFILE declarations. The declarations relate a logical file to a certain physical file that is specified by the OPEN or PREPARE statement performed upon the logical file. The data space used by the declaration holds all of the physical position information needed for that particular file. During file operations, DATASHARE establishes a position within the file using a specified access method and then increments this position based upon the operation specified.

For physically accessed files, a file position is defined by a physical record number (0 through 9693) and a character pointer within this record (1 through 249). When the file is initially opened (with OPEN or PREPARE), the physical record number 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 the 249th character is written during a write, the character pointer is reset to 1 and the physical record number is incremented (when writing, a physical record terminator is automatically written after the 249th data character before the physical record is written out to the disk and movement on to the next physical record is made). If an end of file mark is written, the current physical record is terminated, the physical record number is incremented (unless the position was at the start of a physical record when the operation was entered), the end of file mark is written in the first seven characters of the new physical record, and the character pointer is left

at 1.

The character pointer may be set directly by what is called a tab operation in some disk I/O statements. WRITAB, UPDATE, and all read operations may contain these positioning operators. When physical access is being made to the file, the tab position given in the statement is relative to the beginning of the physical record. When indexed access is being made to the file, the tab position given in the statement is relative to the beginning of the logical record. Note that when tabbing relative to the start of a logical record, it is an illegal operation to tab past the end of a physical record. Therefore, when using tabs in indexed files, there should always be an integral number of logical records per physical record to prevent tabbing past the end of a physical record. Note that tabbing may not be used when physical access is being made to a file declared as indexed. If one needs to do tabbed physical accesses to the file as well as indexed accesses, he must declare two logical files to the same data file. One will be used for physical accesses (having been declared using the FILE directive) and the other will be used for indexed accesses (having been declared using the IFILE directive).

When an indexed file is being used, two additional pointers are kept for the logical file. The first is a physical record number and character pointer to the first character of the last logical record accessed using the index. The second is a pointer to the next sequential key after the last key accessed using the index. The first pointer enables re-reads and updates to be made to the indexed file and the second pointer enables the indexed file to be accessed sequentially by key. Note that neither of these pointers is changed when a physical access is made using the logical file.

An additional counter maintained for all logical files is the space compression counter. This counter is used in the decompression of spaces during read, the compression of spaces during write, and as a flag as to whether or not space compression is to be performed during a write (decompression will always be performed by the read). It is suggested that the reader come back and read the following paragraph closely after he feels he understands the disk read and write access methods and operations since some of these ideas are referenced in the following section. One must understand the following section to be able to effectively deal with space compressed files.

When the space compression counter has a value of -1 during write operations, spaces will not be compressed in the output. The counter value is set to 0 when the file is

initially opened (using OPEN or PREPARE) and at the start of a physically random or indexed access read operation or when a `*+` control in a write operation statement is encountered. The counter value is set to `-1` when a physically random or indexed access write operation is performed or when a `*-` control in a write operation statement is encountered. Therefore, space compression will be on at the beginning of a physically sequential write that occurs as the next operation after the file has been opened or a read operation of any kind has been performed, space compression will be off at the beginning of any physically random or indexed access write operation, and the status of space compression will not be changed by any other operations. If the desired space compression mode for a write operation is not obtained by the above rules then the `*+` and `*-` controls will have to be used to get the desired mode. Note that these controls can erase the memory of previously accumulated spaces if used after the beginning of the statement list while space compression has been on.

#### PHYSICALLY RANDOM ACCESS

The fastest random access method available under DATASHARE is physically random access. To perform a physically random access, a numeric variable containing a positive number is supplied as the record specifier to the statement. Any fractional part of this variable will be discarded and then the physical record number will be set to its value. The character pointer will then be set to one and the read or write operation will proceed. Unfortunately it is often hard to find a map from a key value in the data records to a fairly contiguous set of numbers, necessitating the use of an index structure. However, if such a map can be found, physically random accessing imposes lower overhead than the indexed accessing.

#### PHYSICALLY SEQUENTIAL ACCESS

One can cause the read or write operation to simply pick up where the physical record number and character pointer are currently positioned by specifying a numeric variable with a negative value in the record specifier. Usually, when a read or write operation is finished, it leaves these pointers at the beginning of the next logical record. However, a read or write operation can be parameterized (by placing a semi-colon at the end of the variable list) such that it will simply leave the pointers after the last character dealt with. In this case, the physically sequential access can be used to continue a previous operation from where that operation stopped. The previous operation could have used any access method (including this one) which implies that one can continue a logical record to any length. However, it is often a good

idea to keep logical records reasonably short to prevent them from becoming hard to deal with. Note that the SORT and INDEX utilities require the key value to be within the first 255 characters of a logical record.

#### INDEXED ACCESS

As described in the previous section, a data file may have an associated index file which associates key values to physical record number and character pointer values. There are five basic indexed operations: read a record of a given key value, read a record of the next ASCII sequential key value, update the record that was last accessed through the index, insert a new record of a given unique key value, and delete a record of a given key value. Since there can be any number of indexes into one data file, the insertion and deletion operations will have to perform key insertions and deletions upon all indexes. Therefore, these operations will have to be performed once for each index that points to the data file.

For the indexed read and write operations, once the indexed access has been performed (the physical record number and character pointer values have been set), the actual operation is performed identically to the operation as performed for physical accesses. The one exception is when a record is being inserted. Since records are always inserted at the physical end of the file, a new end of file mark must be written after the inserted record has been written. In this case, a flag is set so that when the write statement has been finished (and it has not been specified that the write operation is to be continued), then the end of file mark will automatically be written. This automatic end of file mark writing operation will not be performed if the write operation is to be continued, thereby making it the responsibility of the DATASHARE program to write the end of file mark when the record has finally been written in its entirety.

The indexed access using a given key value will cause at least one disk sector to be read for each level in the index in addition to whatever disk functions are required to perform the actual read or write operation. If records have been inserted into the index and the INDEX utility has not been run since then, then additional disk sector reads may take place depending upon the length and path of the linked list at the lowest level in the index. Therefore, when many insertions are being performed the INDEX utility should be run as often as is practical to keep the access time from becoming overly large. Also, when a data base is being initialized, it is not a good idea to build it from a null indexed file doing insertions. It would be much more efficient to build the data base physically sequentially as

long as indexed accesses need not be made to it and then create the index file on a reasonably large data file after which additional insertions can then be made using the insertion facility.

#### PHYSICAL ACCESS TO INDEXED FILES

Both physically random and sequential accesses may be made to indexed files. Therefore, one can index only on primary records and then obtain the rest of the records using physically sequential accesses. He may also have a file which is already physically randomly accessed and add an index based on some other key value for fast access to other aspects of the file. If the file has been declared as indexed (using the IFILE directive) then all access methods may be used upon it. However, if the file has been declared as non-indexed (using the FILE directive) then only physical access methods may be used upon it.

#### COMMON FILE CONSIDERATIONS

Since DATASHARE is capable of executing more than one program concurrently, more than one program at a time can try to access a single file. There is no problem if these accesses are not modifying the contents of the file or if they are dealing with different records in the file. If this is the case, one program will have no idea that another is accessing the same file. However, if a certain record in the file is to be modified by more than one program at a time, a lockout mechanism is needed to allow one program to finish its modification before the other can start. The Prevent Interruptions instruction is provided for this purpose (see Section 4.10). The PI instruction can solve many common file update conflicts directly as shown in the example in Section 4.10. However, there are cases where several files may have to be read and then a decision made by the operator before the modification can take place. In this case, the part of the record that is going to be modified can be read first and saved. Then the other reads and operator decisions are made and a new value made ready for the modification write. However, before the modification is actually made, interruptions are prevented while the value currently in the record is read again and compared to the value read the first time. If the value has not changed, then the modification is made before interrupts are allowed again. If the value has changed, a new modification value is computed based upon the new value in the location to be updated (this may require another operator decision) and the cycle is repeated. It is assumed that the conflict rate over a given record in a file will be low and the number of times an operator will be asked to repeat a decision will be small. See Appendix D for a sample program involving modification to a file which could

involve access conflict.

### 7.7.3 PREP or PREPARE

PREPARE (the compiler will also accept a mnemonic of PREP) is used to create a new file under the DOS file structure. The name used for the DOS file name is given in the string variable or literal specified in the PREPARE instruction. The characters used for the name start from under the formpointer of the specified variable and continue until either the logical end of the string has been reached or eight characters have been obtained. (If the item is a literal, the formpointer is one and the logical length points to the last character.) If the end of the string is reached before eight characters are obtained, the rest of the characters are assumed to be spaces. All data files used in DATASHARE are of extension TXT. The character after the 8th in the name variable or the character after the logical length, if the name is less than 8 characters, is used as the drive number for that file. If the character is not an ASCII 0, 1, 2, or 3 or no character physically exists past the name, no drive specification is assumed and all drives starting with drive zero are searched when looking for a name in the directory or directories. Otherwise, only the drive specified is searched.

If a file by the name given already exists (and is not delete or write protected), it is deleted and a new file created. If the file has any protection or the drive specified is off line, an IO error P or M respectively will occur. The logical record number limit is always set to 9695 by the PREPARE instruction.

One always deals with "logical files" in DATASHARE once he has opened them with either the PREPARE or OPEN instructions. Any number of logical files can be opened at one time, the limitation being the amount of space the user has available to devote to the data space needed by each logical file that is declared. The logical files are declared using the FILE or IFILE instructions (see Section 3.1). NOTE: The PREPARE instruction can only create a file that has been declared as a FILE type. The compiler will flag an attempt to PREPARE a file that has been declared as an IFILE type. IFILE type files must be created by use of the INDEX utility running under the DOS.

For example, let the following definitions be made:

FDECL	FILE	
FNAME1	INIT	"FILE1"
FNAME2	INIT	"FILE2 1"
FNAME3	INIT	"ASDFILE32"



Let the formpointer and logical length of FNAME1 be 1 and 5, that of FNAME2 be 1 and 9, and that of FNAME3 be 5 and 9. Then if the statement:

```
PREPARE FDECL,FNAME1
```

were executed, the file FILE1/TXT would be prepared as logical file FDECL on the first drive (beginning with drive 0) on which space was available. If the statement:

```
PREPARE FDECL,FNAME2
```

were executed, the file FILE2/TXT would be prepared as logical file FDECL on drive 1. If the statement:

```
PREPARE FDECL,FNAME3
```

were executed, the file FILE3/TXT would be prepared as logical file FDECL on drive 2. If the statement:

```
PREPARE FDECL,"ASDF"
```

were executed, the file ASDF/TXT would be prepared as logical file FDECL on the first drive on which space was available. If the statement:

```
PREPARE FDECL,"QWER 3"
```

were executed, the file QWER/TXT would be prepared as logical file FDECL on drive 3.

If the logical file specified is already open (having been specified in a previous PREPARE or OPEN instruction and not since in a CLOSE instruction), the old file will be closed before the new one is dealt with.

If the user plans to deal with a vary large file he should run a program that writes a dummy record into the largest record number he plans to use. This will cause the DOS to allocate all records up through the one accessed in as physically contiguous a manner as possible, thus increasing the speed with which the file may be randomly accessed. Note that the use of the DOS implies that a file must be contained on one drive, therefore limiting any one file in DATASHARE to approximately 9500 records (the exact limit depends upon the amount of program information kept on the particular disk). If the writing of the dummy record tries to extend the file past the amount of space available on the disk, an error R will occur.

Remember that space compression mode for writing is left on by a PREPARE instruction (see Section 7.7.2).

## 7.7.4 OPEN

OPEN causes a DOS file already in existence to be prepared for use by the DATASHARE program. Except for the fact that it deals only with files already in existence (giving an IO error if the name specified cannot be found and not killing the file if it already exists), OPEN works in a fashion similar to PREPARE. In addition, OPEN may specify a file that has been declared as an IFILE type (indexed sequential). In the IFILE case, the extension of the name supplied in the literal or string variable is assumed to be ISI instead of TXT (the ISI file header contains the name of the data file it indexes). The opening of the ISI file automatically causes the data file indexed by the ISI file to be opened. If the data file is indexed by more than one index file (ISI file) then each of the indices must be opened using a different logical file for each one. (When dealing with indexed files, the data file itself is never explicitly specified since it is automatically specified by the header of the ISI file that is opened.) For example, if the following logical files were declared:

```
FDECL1    FILE
FDECL2    IFILE
FDECL3    IFILE
```

and a data file FILE1/TXT existed and the ISI files FILE1/ISI and FILE1A/ISI had been created using the INDEX utility as follows:

```
INDEX FILE1;1-5
INDEX FILE1,FILE1A;6-10
```

and the following OPEN statements were executed:

```
OPEN      FDECL1,"FILE1"
OPEN      FDECL2,"FILE1"
OPEN      FDECL3,"FILE1A"
```

then the logical file FDECL1 would be opened to the normal (physical access) file FILE1/TXT, the logical file FDECL2 would be opened to the indexed file whose index name was FILE1/ISI and whose data file name (as specified in the FILE1/ISI header) was FILE1/TXT, and the logical file FDECL3 would be opened to the indexed file whose index name was FILE1A/ISI and whose data file name was FILE1/TXT. This would give physical access plus access via two different indices into the data file FILE1/TXT. Note that an ISI file does not have to reside on the same disk as the data file that it indexes.

Remember that space compression mode for writing is

left on by an OPEN instruction (see Section 7.7.2).

### 7.7.5 CLOSE

CLOSE closes the specified logical file. This insures that any newly allocated space that was not used in the file will be returned to the DOS for allocation to another file.

Example:

```
CLOSE    FDECL
```

If only reads or updates were performed on the file, the CLOSE instruction does not need to be used. Also, a CLOSE is automatically performed when one opens or prepares a logical file that is already open. When a CHAIN is performed, all files that are currently open are automatically closed without space deallocation being performed. Note that this means files cannot be held open across program chains. Also, if the interrupt key is struck or if the port goes off line a chain is automatically invoked meaning that all files will be closed without space deallocation.

CLOSE is also used to delete a file from the DOS file system. If a PREP is performed on a logical file and the next operation performed upon the logical file is a CLOSE, the file described by the logical file declaration will be deleted from the DOS file system.

### 7.7.6 READ

READ performs all file data reads (physically random, physically sequential, indexed random, tabbed or not) except for indexed key sequential reads. The READ statement format consists of a logical file declaration name, a record specifier variable (numeric or string), and a list of variables to be filled by the data from the record. The list may also contain tab indicators which can specify that only certain portions of the data record actually be read into the variables listed. Tabbing is a unique DATASHARE feature which can eliminate unwanted data transfers from and to the disk controller buffer and can allow the programmer to save considerable space in his data area. It can only be used, however, when the logical records do not cross physical disk sector boundaries. This condition can usually be enforced through the use of the REFORMAT utility and careful use of the DATASHARE write instructions.

When data is transferred from the record into a numeric variable that is specified in the READ statement list, the number of characters corresponding to the length of the variable are read in. Any non-leading spaces read will be

converted to zeros (e.g. s3s2s1, where s stands for a space, would be read as s30201). If a non-numeric character other than a negative sign as the first non-space character, decimal point, or space is read, a FORMAT trap will occur. A FORMAT trap will also occur if the variable is dimensioned to one and the character is a negative sign. A FORMAT trap occurs if the data does not match exactly the format of the numeric variable to be read. For example, if X was dimensioned to 4.2 and the characters read were 7777877, a FORMAT trap would occur since the digit 8 appeared where a decimal point appeared in the variable. If a FORMAT trap occurs during a read, the logical file pointers are left pointing at the current file position before the read was attempted.

When a string is read, the number of characters corresponding to the physical length of the variable are read into the variable. The formpointer is set to one and the logical length is set to point to the last physical character in the string.

If the end of the logical record is reached before all variables in the list have been read in full, and the variable which is being filled with data when the EOR is detected is a string, it will have its logical length pointer set to the last character entered before the EOR was reached and the rest of the characters physically in the string padded with spaces. Note that this fact can be used to advantage when reading sequential space compressed files. Remember that the trailing spaces in such file records are not written and that the DISPLAY and PRINT statements can be forced to output only up through the character being pointed to by the logical length (using the `*+` control). These features can be combined to make listing sequential files on the terminal or printer much faster by the deletion of trailing spaces.

The above discussion deals with the action taken when the end of the logical record is reached while reading data into a string variable. If the data is being read into a numeric variable, the rest of the variable is padded with either spaces or zeros as appropriate. Note that if one of these locations within the variable is the decimal point, a FORMAT trap will occur.

If the list contains more variables after the one being filled when the end of the logical record is detected, these variables will either be set to zero (if numeric) or have their logical lengths and formpointers set to zero.

If the list is exhausted before the logical end of the record is reached, two actions can take place. If a semicolon is placed at the end of the list, the file

pointers are simply left after the last character read so a subsequent I/O operation will pick up where the pointers were left. If a semicolon is not placed at the end of the list, the file pointers are advanced until they are pointing after the next logical end of record marker so a subsequent I/O operation will pick up at the start of the next logical record.

A RANGE trap will occur and the logical file pointers will not be changed if an attempt is made to read a record which has never before been written. (Note that the DOS RANGE or FORMAT traps will both cause a DATASHARE RANGE trap and that the DATASHARE FORMAT trap has nothing to do with the DOS FORMAT trap.)

The following is a list of the different types of READ statements. In the examples, the variable RN is a positive numeric item, SEQ is a negative numeric item, KEY is a non-null string item, NULL is a null string item, FNDECL is a FILE declaration name, FIDECL is an IFILE declaration name, and FDECL is either a FILE or IFILE declaration name.

#### TEST FOR END OF FILE

Before discussing the READ operations, the end of file indicator should be discussed. The OVER condition flag being set indicates that a READ operation has run across an end of file mark on physical accesses and has accessed a non-existent key on indexed accesses. The test for the OVER condition should be made after the READ statement. For example:

```
READ FDECL,SEQ;A,B,C
GOTO LABEL IF OVER
```

If an end of file is read on physical accesses, the variables in the statement will be set to zero or have their logical lengths and form pointers set to zero depending upon whether they are numbers or strings respectively. Note that the OVER condition will also be set if a semicolon appeared at the end of the READ list. This is a change from DATASHARE 2 and could require a modification to DATASHARE 2 program logic. The way the READ mechanism works, whenever an end of file mark is found the file pointers "stick" at the beginning of the mark and spaces are supplied for all characters requested to fill variables. Therefore, if one continues to perform READ operations ignoring the fact that the OVER condition flag is being set, the READ operations will simply continue to set the OVER condition flag and clear or zero all variables. This is also true of READ operations whose lists are terminated by semicolons.

The OVER condition being set after an indexed READ

operation indicates that the KEY specified could not be found in the index. For a READKS (read key sequential) operation, the OVER condition being set indicates that the last record in the sequence has been read and the current operation tried to read a non-existent record. See the relevant sections that follow for further information on indexed operations setting the OVER condition flag.

```
READ FDECL,RN;A,B,C
```

This is a physically random access read. The physical record pointer is set to the value of RN and the character pointer is set to the beginning of the physical record (any digits after a decimal point in RN are ignored). Variables A, B, and C are then read. Any remaining characters in the logical record are discarded since the operation leaves the file pointers pointing to the beginning of the following logical record.

```
READ FDECL,RN;A,B,C;
```

This is similar to the above operation except that the file pointers are left pointing to the character after the last one read into the variable C. This enables another I/O operation (write as well as read) to continue from the character after the last one loaded into the variable C.

```
READ FDECL,SEQ;A,B,C
```

This is a physically sequential access read. Variables A, B, and C are read from logical file one beginning at the position indicated by the current file pointer values. The file pointers are left pointing to the beginning of the following logical record.

```
READ FDECL,SEQ;A,B,C;
```

This is similar to the above operation except that the file pointers are left pointing to the character after the last one read into variable C. This enables another I/O operation (write as well as read) to continue from the character after the last one read into the variable C.

```
READ FDECL,ZERO;;
```

Assume that the numeric variable ZERO is defined to be a zero in value. This operation would then cause the file pointers to be positioned to the physical beginning of the file exactly as if a PREPARE or OPEN instruction had just been performed. This implies that

space compression will be on if a WRITE is then performed, and the user must turn off space compression if it is not desired. It is suggested that the reader review the paragraphs in Section 7.7.2, which discuss when space compression is turned on off, if this point is not clear in the reader's mind.

```
READ FNDECL,RN;A,*100,B,*NVAR,C,*50,D;
```

By including the tabbing controls in the read statement list, selected positions may be read from a record without having to read all of the positions in the record. The list controls \*(numeric literal) or \*(numeric variable) are used to position the character pointer to the specified character position in the specified physical record and may appear anywhere in the list. Reading for the rest of the list (unless another positioning control is encountered) begins at the character position specified by the positioning list control. Note that tabbing in physically random access reads is allowed only upon logical files that have been declared using the FILE directive (since the tab values are biased by the starting point of the last index accessed record on reads using a logical file that has been declared using the IFILE directive).

Tab positioning in physically random access read operations is calculated from the first data position of the physical record specified. If the tab position is greater than 249 characters, an IO trap will occur. When reading is completed, the character pointer is moved to the beginning of the next logical record if the statement list is not terminated by a semicolon. If the list is terminated by a semicolon, the character pointer is left pointing one character position past the last character read.

Note that tab positioning in a physically random access read operation will inhibit the ability of that operation to detect an EOF mark that may be in the given sector. Either a non-tabling read can be performed first (to determine whether an EOF exists in the sector in question) followed by the tabbing read if the EOF was not found, or the programmer can invent his own EOF marking convention (which will not require double reads).

The above example would set the physical record pointer to RN and the character pointer to one and variable A would be read. The character pointer would then be set to one hundred and variable B would be read. The character pointer would then be set to the value contained in the numeric variable NVAR and

variable C would be read. The character pointer would finally be set to fifty and variable D would be read. The character pointer would be left pointing after the last character read into variable D since a semicolon appears at the end of the list.

Note that for physically random access reads, it is generally a good idea to place a semicolon at the end of the list if the next read will involve an access to a logical record other than the one which appears next physically. The reason for this is that there is no need to require the processor to scan the rest of the logical record in an attempt to place the file pointers at the beginning of the next logical record when that placement will not be used. This is especially helpful if the read does not leave the character pointer near the end of the logical record as would often be the case where tabbing is being used.

Note that using the read tab on physically sequential access reads (where the record number specified is a negative value) is possible but not advisable. Tab positioning on physical accesses is always calculated from the first character position in the current physical record. The program could obtain characters from a previous or following logical record if tabbing is used in a file where the relationship between logical and physical record boundaries is not known.

```
READ FIDECL,KEY;A,B,C
```

This is an indexed access read. The index file is searched for the key given in the string variable KEY starting with the formpointed character and going through the character pointed to by the logical length. 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 file key items. This means that even if the INDEX utility was told to index on columns 1 through 10, if that field in a certain record consisted of an "A" followed by 9 spaces, the index file key item would consist of an "A" followed by the key terminator character.

If a match is found, the next key pointers are left pointing to the following item in sequence in the index file, the physical record and character pointers are obtained from the index file, and the rest of the



read proceeds precisely as if a physically sequential read were being performed. When finished, the file pointers are left at the start of the physically next logical record in the file.

If no match is found, the OVER condition flag is set, all of the variables in the list are left with the values they had before the READ was attempted, and the next key pointers are left pointing to the next item in sequence in the index file. Therefore, a read key sequential (see the section on READKS) can be performed to obtain the first item by collating sequence following the item that could not be found. This can be very useful for obtaining lists of classes of items.

For example, one could have a file of serialized items with model codes. One could index the file on the model code followed by the serial number. He could then access a given model code with a serial number of all spaces (spaces being lower in collating value than zeros). The access would return with the OVER condition flag set indicating that no such item existed in the file. The program could then proceed to read sequential by key obtaining a list of the serial number of all items of a given model code by the collating sequence of the serial number. The program would have to detect when the model code changed to determine when the list of a given model code should be terminated.

Another feature is that physically sequential accesses can be made after an indexed access. The INDEX utility allows a file to be indexed only upon what are called primary records (this is a SORT utility option). For example, a file could consist of a primary record followed by five secondary records followed by another primary record followed another five secondary records and so forth. If the index were built only on the primary records, one could do an indexed access to the primary record and then do five physically sequential accesses to read the five secondary records.

An indexed access read takes approximately half a second regardless of the size of the data file. This assumes that relatively few insertions have been made upon the file and that only one program is executing in the system. See the section below on index insertions (WRITE) for a discussion on how insertions can affect the indexed access timing.

```
READ FIDECL,KEY;A,B,C;
```

This is similar to the above operation except that

the physical file pointers are left after the last character read rather than at the beginning of the physically next logical record. This is useful if one is not going to do a physically sequential access afterwards since it saves time not scanning to the end of the logical record. It is also useful if one wants to read the rest of the record in a later READ operation or if he wants to update the rest of the record by following the indexed read by a physically sequential write.

```
READ FIDECL, NULL; A, B, C
```

This is an indexed re-read. If the index key supplied to the READ operation is null (logical length and formpointer equal to zero), then instead of accessing a given item based on the key, the operation re-reads the last logical record that was accessed using the index specified by the given logical file. Remember that physical accesses do not change the pointer to the last record accessed using an indexed access. The reader should review Section 7.7.2 if this point is not clear in his mind.

This operation enables one to re-read an indexed record without having to search the index file for a given key. An IO error is given if there has not previously been a successful READ performed using a non-null key on the specified logical file. Otherwise, the operation proceeds exactly as in the normal indexed access READ.

```
READ FIDECL, NULL; A, B, C;
```

This is similar to the above operation except that the physical file pointers are left after the last character read into the variable C.

```
READ FIDECL, KEY; *25, B, *NVAR, C, *10, D;
```

This operation performs an indexed access, positions the character pointer to column 25 relative to the beginning of the logical record, reads the required number of characters into the variable B, positions the character pointer to the column specified in the numeric variable NVAR relative to the beginning of the logical record, reads the required number of characters into the variable C, positions the character pointer to column 10 relative to the beginning of the logical record, reads the required number of characters into the variable D, and leaves the physical record pointers after the last character read. Note the difference between using tabbing in physical accesses

and indexed accesses is that in indexed accesses the tab position specified is made relative to the beginning of the logical record and not to the beginning of the physical record. The reason for this is that one may desire to have several logical records per physical record in an indexed file and be able to use tabbing on the accesses to that file. The problem is that when doing indexed accesses, the program has no idea of where the logical record is in the physical record so the system must make the tab values relative to the beginning of the logical record to make tabbing in indexed files useful. Remember that an attempt to cross a physical record boundary with a tab results in an IO error.

Note that once again it is usually advisable to use a semicolon at the end of statements using tabs since it just wastes time to cause the processor to scan to the beginning of the next logical record if the next access to the file will not be to the physically next logical record.

```
READ FIDECL,NULL;*25,B,*NVAR,C,*10,D;
```

This is similar to the above operation except that the last key-accessed record in the given logical file is read instead of a new index access being made.

#### 7.7.7 READKS

This is a read key sequential operation. As mentioned in Section 7.7.2, whenever an indexed access is made the access routines update a pointer to point to the following key entry in the lowest level of the index. When a READKS operation is performed, instead of searching for a key of a given value, the key pointed to by the next key pointer is used (no key is supplied to the READKS operation). READKS also bumps the pointer to the next key in the index causing successive READKS operations to obtain records in collating sequence. If the pointer to the next key in the index is pointing past the last key in the index (either a key larger than any existing was accessed in the last indexed access or the last key sequential read obtained the last record in the collating sequence) then execution of the READKS operation causes the OVER condition flag to be set and all of the variables in the list will have an indeterminate value. The READKS instruction can appear as follows:

```
READKS    FIDECL;A,B,C
READKS    FIDECL;*25,A,*NVAR,B,*10,C;
```

Except for the access method, the functioning of READKS is identical to the functioning of an indexed access READ (this

is in reference to the action taken once the desired logical record is located).

### 7.7.8 WRITE

The write statement is used for physically random, physically sequential, or indexed insertion writes. The write statement consists of a logical file declaration name and a record specifier (a numeric variable for physical accesses and a string variable for indexed insertions) followed by a list. The list may include variable names, quoted characters, and octal control characters (000 through 037). Note that tab positioning is not allowed in the WRITE operations (the WRITAB operation must be used to do tabbing in writing functions).

Each character string variable in the write list will be written from its first physical character through the logical length. Spaces will be written for any character positions between the logical length pointer and the physical end of string. Each numeric item will be written in total. Note that only the data in each variable is written and not any of the control information (logical length, formpointer, 0200, or ETX). The quoted items and octal control characters will be written exactly as they appear in the list. For example, if the following definitions were made:

TIME	INIT	"10:23"
TOTAL	FORM	"001"
FDECL	FILE	

and the statement:

```
WRITE FDECL,RN;"TIME: ",TIME,015,"TOTAL: ",TOTAL
```

were executed, the file would be written with the characters:

```
TIME: 10:23(015)TOTAL: 001(015)(003)
```

where the (015) and (003) denote control characters. Remember that certain control characters (000, 003, 011, and 015) mean special things to the read operations and their use can cause confusion. In the example above, two logical records were written with the one write statement because of the 015 written in the middle.

The following is a list of the different types of write statements. The variables used are assumed to have the same definitions as those in Section 7.7.6. Although the following examples show lists with only three variables, it should be remembered that all of the WRITE operation lists

can contain the various items shown in the above example.

```
WRITE FDECL,RN;A,B,C
```

This is a physically random access write. The physical record pointer is set to the numeric value contained in RN and the character pointer is set to the beginning of the physical record (any digits after a decimal point in RN are ignored). Variables A, B, and C are then written followed by end of logical record (015) and end of physical record (003) characters. The character pointer is left pointing to the 003 character. Note that all WRITE statements are allowed on either FILE or IFILE declared logical files.

```
WRITE FDECL,RN;A,B,C;
```

This is similar to the above operation except that the 015 and 003 characters are not written after the last data character. The character pointer is left pointing after the last character written. This operation is useful for writing the first part of a record where more of the record will be written later or for updating part of a record where the 015 and 003 would, if they were written, destroy data characters that followed.

```
WRITE FDECL,SEQ;A,B,C
```

This is a physically sequential access write. Variables A, B, and C are written beginning at the character position currently being pointed to by the logical file pointers. If the file had just been opened, the current position would be the first character position in physical record zero of the specified logical file. Otherwise, the file pointers would be positioned according to the results of the last read or write operation executed. End of logical record (015) and end of physical record (003) characters are written after the last character in variable C. The character pointer is left pointing at the 003 character. Remember that space compression mode will be on after the file is opened which means if the file is to be opened and then written sequentially but space compression is not to be used, one must execute a write statement whose first list item is a \*-control. For example:

```
OPEN FDECL,"FILE"  
WRITE FDECL,SEQ;*-,A,B,C
```

See Section 7.7.2 for a discussion of when space compression mode is turned on or off.

```
WRITE FDECL,SEQ;A,B,C;
```

This is similar to the above operation except that the 015 and 003 characters are not written after the last character in the variable C. The character pointer is left pointing after the last character written.

```
WRITE FIDECL,KEY;A,B,C
```

This is an indexed access record insertion. The KEY variable must not be null and the key specified must not already exist in the index specified by the given logical file (either condition will cause an IO error). The search algorithm used to determine that the key is not already in the index is identical to that used in the indexed access READ operation.

The key whose value lies from the formpointer through the logical length of the KEY variable is inserted in the index file specified by the given logical file and the record is written at the physical end of the data file. The record is always started at the beginning of the physical record which contains the EOF mark and then a new EOF mark is automatically written in the physical record which physically follows the new record. Note that this implies that for each record inserted into the data file, at least one physical record will be used (even if the record inserted is only 30 characters long). The record inserted may be longer than one physical record, in which case an integral number of physical records will be used for the inserted record. The reason the inserted record is always started at the beginning of a physical record is that this insures that tabbed operations can then be performed upon the new record in case they are desired (assuming the new record will fit within one physical record).

An insertion takes approximately two seconds to perform on a file which has had no previous insertions performed on it. Insertions can take longer if many records very close together in collating sequence are inserted together. When inserting items whose keys fall randomly within the collating sequence one can usually insert a number of records equal to one tenth of the total number of records in the file before the insertions will start to take significantly longer than two seconds. It is generally a good idea to run the INDEX utility as often as practical when many insertions and deletions are being performed to keep the speed of insertions and indexed accesses as high as possible.

```
WRITE FIDECL,KEY;A,B,C;
```

This operation is also an indexed insertion write except that the new EOF mark is not automatically written at the end of the file. One could desire to finish writing the record with a later operation and could do this by following the above statement by physically sequential write operations and then writing the EOF mark at the end of the file himself. He must make certain, however, that if he is going to do this that no other program can try to do an insertion before the EOF is written or the other program will get a RANGE trap since it will not be able to find the EOF which it will want to overstore.

#### 7.7.9 WRITAB

This operation is the write tab feature which requires a different instruction mnemonic from the normal write operations. With this feature, characters may be written into any character position of a physical record without disturbing the rest of the record. A RANGE trap will occur and the logical file pointers will not be changed if a write tab is used on a record of the file that has never been written before. The write tab can be performed only upon logical files which have been declared using the FILE declaration. The UPDATE operation is used to do tabbed writes into indexed files. The list controls \*(numeric literal) or \*(numeric variable) are used to position the character pointer to the specified character position in the current physical record. Writing of the variable begins at the point specified by the position control. If no positioning is specified, the writing of the first variable starts at the beginning of the physical record.

Tab positioning in physically random accessed writes is calculated from the first position in the specified physical record. If the tab position is greater than 249 characters, an IO trap will occur. Only the quoted characters, octal control characters, and variables appearing in the list are written. The character pointer is left pointing one character past the last character written (there is an implied semicolon at the end of the WRITAB operation). For example,

```
WRITAB FNDECL,RN;A,*70,B,*10,C,*NVAR,"TIME"
```

would write variable A beginning with the first position in the physical record specified by RN. Variable B would be written beginning at position 70 in the physical record and variable C would be written beginning at position 10 in the physical record. The characters "TIME" would be written beginning at the position specified by the numeric variable

NVAR (any places after a decimal point will be ignored) and the character pointer would be left pointing one character past the "E" written for the quoted characters "TIME". An IO trap would occur and the record would not be written if NVAR was greater than 249.

A word of caution is appropriate at this point in the discussion. If in the above example NVAR had had a value of 248, the letter "T" would have been written as the last character in the physical record specified by RN. That physical record would then be written and the following physical record would have been read into the buffer. The letters "IME" would have then been written into the first three positions of this new physical record and the record then written back out. If more tab positions had followed the writing of the characters "TIME", these would have been in the new physical record, not in the one specified by the contents of RN. This action would probably not be that expected by the programmer and would all take place without a wimper of an error message from the interpreter. Just be careful about your tab positions!

Note that using WRITAB with a physically sequential access (where RN contains a negative value) is possible but not advisable. Tab positioning on physical accesses is always calculated from the first character position in the current physical record. The program could obtain characters from a previous or following logical record if tabbing is used in a file where the relationship between logical and physical record boundaries is not known.

#### 7.7.10 UPDATE

This operation allows modification of the last record that was accessed with a READ or READKS operation. Only the logical file declaration name is supplied to this operation (no key is supplied) but the list may have all of the items allowed in the WRITAB list. For example,

```
UPDATE FIDECL;A,*20,B,*40,"ASDF",033
```

would read the last indexed accessed record in the logical file FIDECL and would overstore the first characters in the logical record with the contents of the variable A, would overstore the characters starting with the 20th character in the logical record with the contents of the variable B, and would overstore the characters starting with the 40th character in the logical record with the characters "ASDF" followed by the octal character 033. The character pointer would be left pointing after the 033 character (the last character written from the list). Note that as in indexed access reads using tab positioning, the tab positions in the UPDATE operation are relative to the beginning of the



logical record (and not the beginning of the physical record as in WRITAB). As in the WRITAB operation, the UPDATE has an implied semicolon at the end of its list.

#### 7.7.11. INSERT

This operation allows an index insertion into more than one index file. The WRITE operation mentioned earlier is used to physically insert the record into the data file and insert the key into one index file. If more than one index is being used, one INSERT operation must be performed for each additional index into which an insertion is to be made. When the WRITE operation performs the physical record insertion, a pointer is kept which contains the physical location of the newly inserted record in the data file. When the INSERT operation is performed, the specified key (with a pointer to the remembered physical location into the data file) is inserted into the specified index file. Since only one of these insertion memory pointers is kept for each program, one must make sure that he performs all insertions necessary for a given record before performing the next WRITE to insert the next record. For example, the sequence to insert two records into two indices would be WRITE INSERT WRITE INSERT and not WRITE WRITE INSERT INSERT. The format of the INSERT statement is as follows:

```
INSERT FIDECL,KEY
```

where FIDECL is the name of the logical file declared for the index being used and KEY is the string variable in which from the formpointer through the logical length is contained the key to be inserted in the index. An IO error is given if KEY is null or if the key specified already exists in the specified index file. Otherwise, the key is simply inserted into the index. Note that it is not necessary to prevent the program from being interrupted between the WRITE and INSERT operations since the pointer to the record which was inserted is kept for each program and even if another program inserted a record in the same file or index between the WRITE and INSERT of the program in question, all insertions would be performed correctly.

#### 7.7.12 DELETE

This operation 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 other indices which can index the data file. For example,

```
DELETE FIDECL,KEY
```

will delete the record specified by the key (whose value

lies from the formpointer through the logical length in the variable KEY) in the data file specified by the index file specified by the logical file whose declaration name is FIDECL. The record is physically deleted by having all of its characters up through the logical end of record mark (015 character) overstored with 032 control characters. The 032 character does not appear to exist when the record is read using the DATASHARE read mechanism or the REFORMAT utility read mechanism since when these mechanisms see such a character they simply bump the character pointer (moving on to the next physical record if running off the end of the current physical record) and try to fetch the next character. Therefore, when DATASHARE performs physically sequential reads across records that have been physically deleted, the records do not appear to exist. The REFORMAT utility eliminates these 032 characters to close up the deleted space in a file and to make the file readable by other DOS utility programs such as SORT.

The DELETE operation will not try to overstore the record being deleted with 032 characters if the first character already contains a 032 character. This allows the DELETE operation to be used to delete the key entries from all index files which index the given data file. For example,

```
DELETE FIDECL1,KEY1
DELETE FIDECL2,KEY2
DELETE FIDECL3,KEY3
```

would be used to delete the record and keys out of the three indices which pointed to that record. The first DELETE would actually overstore the logical record with 032 characters and delete the key from the index file specified by the logical file whose declaration name was FIDECL1. The other two DELETE operations would only remove the keys from their respective index files since it would be noted that the logical record already contained a 032 character in its first position.

#### 7.7.13 WEOF

Standard DOS end of file marks (000 000 000 000 000 000 003) in the first seven character positions of a physical record) can be written in DATASHARE. WEOF does not change the physical record or character pointers for the given logical file. For example,

```
WEOF FDECL,RN
```

will write an end of file mark in physical record RN while

```
WEOF FDECL,SEQ
```

will write an end of file mark in the next physical record after the current physical record pointer. Note that the WEOF operation may be performed upon logical records which have been declared either FILE or IFILE but that the record is always specified using a numeric variable for the record number. This implies that one cannot write an end of file mark using an indexed access.

## 8. PROGRAM GENERATION

### 8.1 Preparing Source Files

Files containing the source language for DATASHARE 3 programs are prepared using the general purpose editor running under the DOS and whose use is covered in a separate document. The editor tab stops may be set to be suitable for keyin of DATASHARE 3 programs by using the :T command and setting two tabs, one at 10 and the other at 20.

### 8.2 Compiling Source Files

DATASHARE 3 programs are compiled using the DATASHARE 3 compiler running under the DOS. Note that DATASHARE 3 programs must always be compiled using the DATASHARE 3 compiler running alone under the DOS. This means that programs cannot be generated while the DATASHARE 3 system itself is running, and if it is running it must be stopped either manually or with the use of ROLLOUT before compilations can be performed. The DATASHARE 3 compiler is parameterized in the following manner:

```
DSCMP <source>[,<object>][,<print>][;<L><C><E><R><X><D>]
```

#### File Specifications:

The compiler may be parameterized with up to three file specifications. These file specifications follow the standard DOS conventions. Refer to the DOS User's Guide for further information concerning DOS file specifications. A bad drive specification for any of the files will result in the error message:

BAD DEVICE SPECIFICATION

If any of the file specifications are identical, the message:

```
SOURCE AND OBJECT FILES THE SAME    or
SOURCE AND PRINT FILES THE SAME     or
OBJECT AND PRINT FILES THE SAME
```

will be displayed.

The source file contains the DATASHARE 3 program text created with the editor. This file must always be specified. If no extension is given on the source file name, the extension TXT is assumed. If the source file name is not supplied, the message:

NAME REQUIRED.

will be displayed. If the source file name does not exist in the DOS directory, the message:

NO SUCH NAME.

will be displayed. If no drive is specified, all drives beginning with drive 0 will be searched for the source file.

The object file will contain the object code generated by the compiler from the specified source code. If the name of the object code file is not given, the name of the source code file with an extension of DS3 will be used for the name of the object code file. Note that DATASHARE 3 can run only those files with extension DS3. If the source code file is specified without a drive number, the compiler will search all drives for the name given. If the object code file name (with the extension specified or the assumed extension DS3) is not found on any drive, the object code file is placed on the same drive as the source code file. If the object code file is found, it is killed and re-opened on the same drive it was found on to assure a maximally contiguous file space is available. This will lessen the likelihood of the occurrence of a segment error (explained later).

The print file specification is also optional. If it is given, any print output requested will be written in this file (in the standard GEDIT format) instead of being printed on the local printer. Top of form will be indicated by the character '1' in column one of the print line. Otherwise, column one is always blank and the line starts with column two (this is the standard COBOL and FORTRAN print file format). This option is particularly useful for compilations during ROLLOUTs (see Section 4.9). For example, during the ROLLOUT several compilations could be run which placed the print output into the print files specified. The compilation results could then be printed by a DATASHARE 3 program when the DATASHARE 3 system was restored. This procedure would shorten the total time that the DATASHARE 3 system would have to be down while at the same time allowing the programmer to obtain program listings.

If no name is given for the print file specification, the source file name will be assumed. If no extension is given, an extension of PRT will be assumed. However, if the print file is to be read under DATASHARE 3 it must have an extension of TXT since all DATASHARE 3 data files must have that extension. If no drive number is specified, the print file will be placed on the same drive as the source file. A print file may be specified simply by keying in a comma after the object file specification or, if no object file is specified, by keying in two commas after the source file specification. Note, however, that the extension assumed in

this case will be PRT.

#### Output Parameters:

These parameters allow the user to specify what type of output is wanted in addition to the object file. If a print file is specified, any print output is written in that file instead of being sent to the printer. If the semicolon but no parameters are specified, the only output is the object file (if in this case a print file was specified it would be null).

The DATASHARE 3 compiler can output to either a local or servo printer. The compiler is self-configuring in this respect and will output to whichever printer it finds connected to the system I/O bus. Since the compiler looks first for a servo printer, output will be to the servo printer if both a local and servo printer are addressable by the system.

Any source code lines which have errors are displayed on the screen during pass II, with the appropriate error flag. Additionally, the compiler displays at the lower left corner of the screen the current line number being compiled, for every 10th line. Every 10th line is indicated because displaying the line number for every line would slow down the compiler. No numbers will be displayed if the program is fewer than 10 lines long. This line number display is cleared when processing of included files begins or ends, so the line number display will blink off momentarily during compilation of source files using included files.

To specify output options, a semicolon plus one or more of the following should be placed after the last file specification:

- L A listing of the compilation results is printed. Each line of source code is numbered and the object code location counter value for the first byte of code generated for the line is listed to the left of each source code line. A '+' appearing as the first character of a line causes a new print page to be started. The rest of the line following the + may be used as a comment line. A '\*' appearing as the first character of a line causes a new print page to be started if the current line is within two inches of the bottom of the current page. A good way to improve the readability of a program is to begin each section or routine with a comment before which a line is entered which contains a star in its first column. This will make sure the comment appears on the same page as the first lines of the code to which it is attached.

- C A listing of the compilation results is printed and the generated object code is listed to the left of the source code. Printing the object code usually makes the listing about twice as long. If this option is given, the L option is implied and need not also be given.
- E The source code for lines with errors will be printed in addition to being displayed on the screen. This parameter has no meaning if the L or C options are given since listings produced under those options will include error flags anyway.
- R The line numbers for referenced labels in an operand string will be printed at the right margin of the listing. The line number is the line on which the Referenced label was defined. If the L, C, or E option is not also given, this option has no effect. This option may be given instead of or in addition to the X option. The R option is especially convenient with GOTO or CALL instructions in following the logic path of a complex set of code. Note that for the R option to be effective, a printer with at least 130 column printing capability must be used.
- X A cross-reference listing is printed at the end of the compilation. There will actually be two cross-references: one for the data labels and one for the executable labels. Each cross-reference is sorted alphabetically. The data or executable label is given preceded by the octal location where the label was defined and followed by a list of all line numbers in which the item was defined or referenced. An asterisk flags those line numbers which are definitions. The SORT utility is called by the compiler to do the actual reference sorting, and the messages displayed on the screen will be appropriate to the progress of the sort. A cross-reference may be obtained regardless of whether a listing was requested.
- D A copy of the source code is displayed on the screen during the compilation.

If a listing has been requested, the compiler will ask:

HEADING:

This may be 70 characters long and is printed at the top of each page. Indicating the time and date of the listing is helpful in keeping listings in chronological order. The source file name is automatically listed to the left of the heading.

Examples:

DSCMP PROGRAM

This is the simplest compilation specification. The source code found in file PROGRAM/TXT would be compiled with the object code placed in file PROGRAM/DS3. No other output would be given except for errors displayed on the screen.

DSCMP ANSWER,ANSWER4;CX

The source code in ANSWER/TXT would be compiled and the object code placed in ANSWER4/DS3. A listing would be printed on the printer and consist of the source and object code with a data and executable label cross-reference at the end.

DSCMP FILE:DR0,,FILELST/TXT:DR1;LX

The source code in FILE/TXT on drive 0 would be compiled and the object code placed in FILE/DS3 on drive 0. A copy of the source code and a data and label cross-reference will be written in FILELST/TXT on drive 1.

The compiler may be stopped temporarily by depressing the DISPLAY key. The DISPLAY light will be turned on and execution will not be resumed until the DISPLAY key is depressed again (the DISPLAY light will then be turned off). Compilation may be aborted at any time before the cross-reference sort is begun by depressing the KEYBOARD key. If the compilation is aborted in this manner the object file and the dictionary file are deleted, as are the reference file and the print file if a cross-reference list or print file was specified.

### 8.3 Compilation directives

Two directives are available in the DATASHARE 3 compiler as mentioned in Section 2.2. One is the EQU statement which allows a label to be assigned a decimal numeric value from 1 through 249. For example:

```
LM      EQU      5
```

A label which is defined in this manner may be used as tab values in disk I/O statements and as cursor positions in KEYIN, DISPLAY, and CONSOLE statements. This is particularly useful when one defines a data base record format. If all item positions within the record are defined using the EQU directive, then changes in item positions can be achieved by simply changing the one directive value. If the EQU were not used, the user would have to hunt through all programs to change all disk I/O statements to change the



item position in the record.

The second compiler directive is INCLUDE (the compiler also accepts a mnemonic of INC) which allows another text file to be included at that point as if the lines actually existed in the main file. For example:

```
INC      RECDEFS
```

will cause the file RECDEFS/TXT to be scanned as if all of its lines existed in the place of the INCLUDE line. The assumed extension on included files is TXT but may be specified to be any extension. If no drive is specified, all drives starting with drive zero will be scanned for the file. Inclusions may be nested up to four deep. The INCLUDE directive can be used to include a file containing the EQU directives and data variable definitions which define the format of a data base file record. This can prevent the programmer from having to keyin the data area (and common data area) definitions over and over for each program to use a certain data file. It also will make it much easier to update the data area definition since the programmer would have simply to update the one text file and then compile all the programs (which would include the modified definition file) to update all programs to the new data area definition.

#### 8.4 Compilation diagnostics

The compiler prints and displays diagnostic messages on the listing to help the programmer debug syntactical errors in his code. These messages take the form of an error code letter at the left of the listing and an asterisk under the line at the position of the scanning pointer when the error occurred. The letters are E for an expression error (a generalized syntactical error), U for an undefined variable or label, and I for an undefined instruction. In the case of E errors a number is given on the line with the asterisk pointing out the error position in the source line. This number refers to the list of detailed error explanations in Appendix E of this document. If any of these flags appear, the compiler will store a STOP instruction into the first executable location in the object file. If the faulty program is then executed, it will only execute the STOP instruction which will simply return control to the MASTER program.

The DATASHARE system uses the DOS logical file zero for reading and writing all data to and from the disk. This implies that a segment boundary may not be crossed by the object code during a READ or WRITE statement (since fetching the statement also involves disk I/O). For this reason, DATASHARE object files are restricted to one segment in

length. If, during code generation, more than one segment was used to hold the object file, the compiler gives an error message:

#### SEGMENT ERROR

and flags the file by storing a STOP instruction into the first executable location in the object code file. In this case, the object code file for the given program should be deleted from the DOS (using the KILL command) and the disk pack onto which the object code was being written should be purged or the BACKUP program used (if a dual drive system is available) to make more contiguous free space available. To minimize the possibility of a segment error if the object code file already exists (since the program could become larger), the compiler kills the old object file and then re-opens it on the same drive to assure a maximum of contiguous file space.

#### 8.5 Disk space requirements

The DATASHARE 3 compiler maintains its label dictionary on disk in the file named DSCDICT/SYS. Moreover, this file is always placed on the same drive as the output object file because it is reasonably certain that that drive will not be write protected. For these reasons, there may not be more than 254 files named (255 if the object file name already exists) on the disk onto which the object file is to be written.

Further, if a cross reference is desired, there must be four more file name places available among the drives on-line. One of the file names that will be in use during the compilation is DSCREF/SYS (the file onto which the compiler writes information about each label reference). Three files will be generated by SORT: \*SORTMRG/SYS, \*SORTKEY/SYS, and DSCREFT/SYS. The first of the two files by SORT are scratch files, and the third is a tag-file pointing back into the DSCREF/SYS file. At normal completion of the compilation, all files mentioned above (except the output object file) will have been deleted and the file space again made available to the user.

## 9. SYSTEM GENERATION

### 9.1 Loading From Cassette

The DATASHARE compiler and interpreter system programs are contained on one cassette. The cassette is in the DMF (DOS Multiple File) format which includes a directory of the files on the tape. All that is necessary to load the DATASHARE 3 system files to disk is to have the MIN program catalogued on the system and to keyin:

```
MIN ;A
```

The MIN (Multiple IN) program will be activated and will display the date of creation of the tape, the file names in the tape directory, and each file name as the file is being loaded. If the file already exists on the disk, the MIN program will ask if it is to be overstored. The operator can decide to overstore the file or can tell it not to overstore the file in which case MIN will allow the file to be stored under a different name. Consult the MINMOUT USER'S GUIDE for further information on its operation.

The files contained on the DATASHARE 3 release tape are the interpreter system (DS/CMD and DS/OV1 through DS/OV7), the compiler system (DSCMP/CMD and DSCMP/OV0 through DSCMP/OV2), and the other support programs (DSCON/CMD, DSBACK/CMD, DSBACKTD/CMD, AUTOKEY/CMD, ROLLOUT/SYS, ROLLFILE/SYS, and INTRHAND/SYS).

The DATASHARE 3 interpreter system files can be re-named to any name desired as long as the command file and all the overlays have the same name. For example, if DS/CMD was re-named DS3/CMD, then DS/OV1 thru DS/OV7 would have to be named DS3/OV1 thru DS3/OV7.

### 9.2 Port Configuration

The DATASHARE system may be configured to run with from one to eight ports. The system is configured by running the DSCON program. This program will first display the current configuration (if one has been made) and then ask if the configuration is to be changed. If a negative response is given, control is returned to the DOS. Otherwise, the DSCON program will run through a sequence of questions concerning the number of ports, whether the console is to be the terminal for port one, whether the servo printer is to be used for the system printer, if port one is on the console whether the multi-port is to be bypassed altogether, and whether the available space is to be divided evenly among the ports. If the space is to be evenly divided, the DSCON program will display how much space is allocated to each port. If the space is not to be evenly divided, the DSCON

program will request the amount of space to be allocated to each port. The amount of space must be at least 20 bytes and may never be more than the total amount of space left.

### 9.3 Necessary Programs

Before the DATASHARE system can be used, two more sets of programs must exist. These are called the ANSWER and MASTER programs and perform the tasks of dealing with the user when he initially signs onto the system and dealing with him when he is not running another DATASHARE program. Note that all execution in the DATASHARE system occurs in the high level language and since the user writes his own ANSWER and MASTER programs, he can determine how the system command language appears. The ANSWER and MASTER programming concepts are dealt with in Section 11.

If an ANSWER and MASTER program do not exist for a port, it will never become active even if it is configured into the system. The ANSWER and MASTER program must have the object names ANSWERn/DS3 and MASTERn/DS3 where n is the number of the port for which these are the ANSWER and MASTER programs (n = 1 thru 8). All ANSWER and MASTER object files must reside upon drive 0 in the system. If a multi-drive system is being used, it is generally a good idea to keep all necessary system utilities, the DATASHARE system files, and the DATASHARE object code files, as well as the ANSWER and MASTER program object files on drive 0 and to not remove the disk in drive 0 during normal system operation.

Other programs which should be on the system include the INDEX, REFORMAT, and SORT utilities for the generation of index files. Also, the MINMOUT utilities should be on the system as they would be the programs used to dump and reload DATASHARE object code files to and from cassette tape.

## 10. SYSTEM OPERATION

## 10.1 Bringing Up the System

If the DATASHARE 3 interpreter system files are named DS/CMD and DS/OV1 thru DS/OV7, then the DATASHARE system is brought up by entering the DOS command:

DS

This begins a series of operations the first of which is the display of the message:

DOS 1.2 DATASHARE 3.r - SYSTEM BEING INITIALIZED

where r is the revision number of the particular release. If the DATASHARE system has not been configured, the message:

\* DATASHARE 3.r HAS NOT BEEN CONFIGURED \*

is displayed. If the configuration file cannot be found on the same drive the DS/CMD file is located on, the message:

\* DSCON/CMD MISSING ON DRIVE d \*

is displayed where d is the drive number 0 through 3. If one of the constituent overlays of the DATASHARE 3 interpreter system cannot be found on the same drive the DS/CMD file is located on, the message:

\* DS/OVn MISSING ON DRIVE d \*

is displayed where n is the overlay number (1 through 7). Of course, if the DATASHARE 3 interpreter system files have been re-named, the names in the above messages would be changed accordingly. If any of the above messages after the initialization message is displayed, the machine will beep and halt. If the auto-restart tab is punched on a rear cassette containing the DOS bootstrap loader, then this action will cause control to return to the DOS. Otherwise, the RESTART key must be pressed to cause the machine to run again. If the initialization is completed successfully, the system displays the message:

OPERATOR, PLEASE DEPRESS THE KEYBOARD OR DISPLAY KEY.

This action will verify that an operator is present. A design objective was that the time and date be initialized by the operator when the system was brought up but that the system also be capable of bringing itself up in the case of power failure and unattended operation. If the keyboard or display key is not depressed within 30 seconds after the

message is displayed, the machine will make a series of one second beeps in an effort to attract the attention of any operational personnel within the vicinity. If the keyboard or display key is not depressed after 30 seconds of beeping, the system assumes that it is being operated in an unattended mode and should start operation without the time and date being initialized. In this case, the time and date entries at the upper right of the 2200 screen will be blank.

If the time and date are to be initialized, the operator must depress either the keyboard or display key. Upon doing this, the screen will be initialized with a message indicating the release of the DATASHARE system being used, the number of ports configured for that system, and the digits one through eight running down the left side of the screen. These digits denote a line which is allocated for each physical port. The CHAIN statement displays on this line the name of the program being invoked. The program running for that port may also display on this line using the CONSOLE statement. These lines are useful for informing any operational personnel of the status of the system.

To initialize the time and date, the system will display the message TIME: in the upper right part of the screen. The operator should respond to this with a four digit number indicating the current clock value in hours and minutes (HHMM). Note that no colons should be entered and that a valid 24-hour clock value must be entered. If the value is not valid, the TIME: message will be repeated. Otherwise, the system will display the message DATE: to the right of the time value just entered. The operator should respond to this with a three digit number followed by a slash followed by a two digit number. The first number should be the current julian date (a number between 1 and 365 or, on leap years, 366) and the second number should be the last two digits of the current year. Note that the format mentioned must always be followed, with leading zeros used if necessary. If the julian date is not valid, the DATE: message will be repeated. Otherwise, the system will begin execution as denoted by the wall clock display running in the upper right part of the screen. A period of approximately 20 seconds will pass while the system looks up all of the ANSWER and MASTER program names in the DOS directory and stores their physical file numbers away in a table. Ports requesting connection during this time will be connected but no response will be made until the 20 second period has passed. Note that an asterisk just to the right of the port number at the left side of the screen will be displayed if the Carrier Detect signal for that port is present.

If the system is configured to run port one on the

console, an alternate form of bring up the system may be used. Instead of entering the simple DOS command DS to start the system, the operator can enter:

DS <program>

where <program> is the name of a DATASHARE object code file on drive zero. If this action is taken, the file <program>/DS3 will be used for the answer program for port one instead of ANSWER1/DS3. The master program for port one will still be MASTER1/DS3. Also, the operator will not be requested to depress the KEYBOARD or DISPLAY keys and to enter the time or date. The time and date will be initialized to 00:00 and 000/00 respectively and execution will start immediately. Note that the console screen is blanked just before execution is begun if port one is being run on the system console. If one wishes to simply bypass the time and date entry, he can enter the command:

DS ANSWER1

which will allow the normal answer program to be executed for port one but will eliminate the request for the time and date. This feature makes it possible to run DATASHARE from the CHAIN program. To return from DATASHARE to the chaining process, a ROLLOUT must be performed with the DOS command:

CHAIN #

given which will cause the CHAIN command to pick up after the last command issued.

The DATASHARE 3 interpreter system can be caused to automatically execute when the DOS is brought up by the use of the AUTOKEY/CMD program. DATASHARE 3 has been changed over DATASHARE 2 to look in the DOS command line to determine the name of the command file so the overlay names can be determined (this is what allows the DATASHARE 3 interpreter system files to be re-named). Because of this, the standard DOS AUTO program can no longer be used to directly cause automatic execution of the DATASHARE 3 interpreter system. There are a number of DOS programs which use the same technique of looking into the DOS command line so a general solution to the problem of not being able to automatically execute these programs was implemented.

The AUTOKEY program has two modes of execution. Upon starting execution AUTOKEY sees if the KEYBOARD key is depressed. If so, AUTOKEY will request a line to be entered and then store this line into a sector in the AUTOKEY/CMD file which follows the end of file mark in the object code. The line entered may be any single line DOS command. If upon starting execution AUTOKEY does not see the KEYBOARD

key dispressed, it will read the line it had previously stored in its own object code file into the DOS command line area and then enter the DOS in a place which will cause the line to be interpreted as if the line had been entered from the keyboard. To cause the DATASHARE 3 interpreter system to be automatically executed upon loading of the DOS, one would run AUTOKEY holding down the KEYBOARD key and when the line were requested enter "DS". He would then run the DOS AUTO program by entering "AUTO AUTOKEY". The next time DOS was brought up, the AUTO function of DOS would load and execute AUTOKEY which would execute the DOS command "DS" as if it had been entered from the keyboard.

## 10.2 Taking Down the System

The DATASHARE system maintains its files totally under the control of the DOS. The DOS normally may be halted at any time without detriment to the file structure. However, halting the system after a new file has been created or after a new segment has been allocated will leave that file with the maximum amount of space allocated to it. Proper closing of the file collapses the space allocated to only that used. Thus, to be sure all files are properly closed, the system should be halted when all ports are in their MASTER programs. The operator can tell from the console screen when a port is in its MASTER program if the MASTER program displays its name as in the examples in Appendix C.

## 10.3 Fatal Error Conditions

There are error conditions within the DOS which cannot be trapped. These errors invoke a DOS overlay called the ABORT overlay which reloads the DOS to insure the presence of the DSPLY\$ routine, displays an error message in the standard DOS format, and then returns control to the DOS command interpreter. Note that this sequence does not provide for restoring the foreground interrupt handler or insuring that the DOS does not overlay an interrupt process that happens to be running. The DATASHARE foreground routines reside in an area which is overlaid by the DOS and, therefore, the normal abort message routine would cause havoc when it tried to load the DOS. For this reason, the DATASHARE system overlays the DOS in a critical place that allows it to trap the action of untrappable DOS errors and store a return instruction in location zero. This effectively disables any interrupt handler execution and allows the DOS to be loaded for the abort message display but does not restore the normal DOS foreground interrupt handler. The DATASHARE system also overlays the DOS EXIT\$ entry point with a jump to a beep and halt. This causes the machine to halt when the untrappable error message display is completed.



## 11. ANSWER AND MASTER CONCEPTS

There are two DATABUS programs which must exist for each port for that port to be active. The first is called the ANSWER program and must have a name of ANSWERn where n is the number of the port. For example, ANSWER1 for the first port, ANSWER2 for the second, and so on. The ANSWER program deals with the user when he initially connects to the system (calls on the telephone or turns on his CRT). The second program is called the MASTER program and must have a name of MASTERN where n is the number of the port. The MASTER program deals with the user whenever he is not executing the ANSWER program or an application program and is generally used to allow the user to select the next application program he wishes to execute. Note that both of these programs are written in DATABUS, enabling the user to tailor the command aspects of the DATASHARE system to his particular needs. Simple and complex examples of ANSWER and MASTER programs are shown in the appendices. Remember that the object code for all ANSWER and MASTER programs must reside on drive 0.

### 11.1 System Security

The ANSWER program allows the programmer to force the user to give some type of identification before he is allowed to use the system. Note that the INTERRUPT key on the terminal is ignored while execution is taking place between the time when the system first acknowledges the presence of a user at a given port and the first chain executed by the program for that port. This means that while the user is executing in the ANSWER program for a given port when he first signs onto the system, he may not escape around the identification request and get directly into the MASTER program by simply striking the INTERRUPT key. The ANSWER program may also be structured to enforce file access limitations depending upon the identification of the user.

### 11.2 System Convenience

The ANSWER program chains to the MASTER program which usually requests from the terminal operator the name of the program he wishes to execute. This name can be generated from information supplied by the terminal operator so, for example, the operator may enter the number of a form and the MASTER program will decide which program to execute for that form number. The DOS directory cannot be directly accessed by the MASTER program, implying that a file must be generated which contains the names of programs and files that are to be accessed if directory service or file access limitation is to be implemented. It is very much up to the author of the ANSWER and MASTER programs to provide any

convenience facilities to the terminal user.

### 11.3 Sample Answer and Master Programs

Appendix C contains examples of both simple and complex ANSWER and MASTER programs. Each program is edited for entry of the appropriate port number in the variable PORTN and then compiled for the given port. This procedure (editing in the port number and then compiling into an object file with the port number in its name) must be followed for each port that is to be used in the system. If a DATASHARE object file for either the ANSWER or MASTER program does not exist on drive 0 for a given port, the port will simply not be activated when the system is brought up.

The simple ANSWER program displays on the terminal the number of the port and displays its program name on the console. The latter action is performed because the system does not display the name of the program invoked when the chain was caused by action other than the execution of a CHAIN statement (e.g., the ANSWER program initiated by terminal connection or the MASTER program initiated by a STOP or INTERRUPT key). The system does display on the console line allocated for the executing port the name of all programs invoked by the CHAIN statement. The simple ANSWER program then requests an identification and checks it for validity against a very simple rule (the identification given must be exactly the word DATAPOINT). If the word matches (note the use of both the NOT EQUAL and LESS conditions for checking for an exact match), a STOP statement is executed which causes a chain to the MASTER program. Otherwise, an indication is given that the proper identification was not entered and another request for identification is made.

The simple MASTER program merely requests the name of a program to be executed. A CHAIN is executed to the name given and if a chain failure occurs an indication is given that the name does not exist in the DOS directory and another request for a program name is made. Note that both the ANSWER and MASTER programs are written without the use of cursor positioning in the KEYIN and DISPLAY statements to aid in Teletype terminal compatibility.

The MASTER program should not assume any common data areas since it can be entered due to a program trap or the INT key being struck. For this reason, if a common data area value is to be determined (such as the port number) this should be done in the MASTER program and not in the ANSWER program.

The complex ANSWER and MASTER programs perform tasks similar to those performed by the simple programs except

that a number of convenience features are added to give the system the appearance of a more conventional time sharing system. Two files are associated with the more complex programs, the SYSFILE and the DAYFILE (system and day files). The system file contains identification code information and a table associating a given identification code (user) with a given set of programs (user's directory). The system file also contains a record for each physical port (records zero through seven) which allows any executing program to determine which user identification is associated with the given physical port at any given time. A user identification number (an index into the rest of the file from which the actual symbolic user identification can be obtained), the time at sign on, and the date at sign on are recorded in this record. The remainder of the file contains four records for each user identified in the system. Each record is broken into ten ten-character fields. The first field of the first record is the identification code. The rest of the fields in the first record and the following three records contain program names associated with the given user identification. The list of program names is terminated by a space appearing in the first column of the name. The list of user identifications is terminated by a space appearing in the first column of a user identification.

The second file associated with the complex ANSWER and MASTER programs is called the day file. This file simply contains a set of records to be displayed at sign on time. This information is used to inform users of changes in the system or any other facts pertinent to the use of the system. Note that both of these files must exist before the complex ANSWER and MASTER programs can be used. The files can be created with DATASHARE if simple ANSWER and MASTER programs exist.

The complex ANSWER program determines the month and day of the month from the julian date. It detects if the date has not been initialized by noting that the julian date is zero (an invalid initialization value). After the date is displayed, a request is made for an identification code. The identification code list in the system file is then scanned for a match with the one supplied. If a match cannot be found, an indication is given to the user and the request for identification is repeated. Note that only three tries at identification are allowed in an effort to prevent unauthorized access to the system via the technique of trying identification codes until one is struck. After the third try, the response to the user does not change but he is not allowed access to the system even if he does then enter a valid identification and an alert message is displayed on the console to alert the operator that someone who apparently does not know an identification code is

trying to access the system. If a valid identification is entered within three tries, the identification index into the system file, the date of sign on, and the time of sign on are written in the record in the system file corresponding to the physical port being used and execution is passed to the MASTER program via the STOP statement.

The complex MASTER program allows a number of commands as explained in the KEYIN statement under the label HELPI. This particular program does not limit program or file access to a given user to his programs only, but such a scheme could be implemented without much difficulty.

Note that when the ANSWER program is chained to it will execute until the first KEYIN, DISPLAY, or CONSOLE statement is executed. The ANSWER program is actually executed when the terminal disconnects from the system, not when it connects to it. If the time of connection and disconnection and total connection time are being kept in a file, the ANSWER program can note when a user disconnects from the system and log the total amount of time the user was connected as the first operations in the ANSWER program. Then the KEYIN statement requesting a new user identification can be issued which will cause execution to cease for that port. The log out function will be executed when the terminal disconnects from the system. When the terminal re-connects to the system the KEYIN statement will be satisfied when the operator at the terminal enters an identification code at which time the new user can then be logged on with the time being noted in the log file. Note that when the system is initialized, all ports will appear to be logging off (since all ANSWER programs are executed) but no corresponding log on time will be set. The program must handle this special case by allowing for log offs without corresponding log on times.

## 12. PHYSICAL SYSTEM CHARACTERISTICS

### 12.1 Virtual Memory

To achieve a reasonable amount of program space for eight simultaneous programs, DATASHARE employs a virtual memory technique. DATASHARE code is very compact, with very few bytes of instructions being capable of invoking a large amount of processor activity. Therefore, the rate at which DATASHARE program bytes are fetched is very low. Because of this low rate, the actual program code bytes can be kept in the randomly accessible disk buffers with very little effect on program execution speed. Three of the four disk buffers are used for the storage of pages of program code. This gives the effect of having a DMA channel from the disk to the high speed program storage memory. Another characteristic of DATASHARE code is that it is never modified. Because of this, program code need only be read in and never written back out to the disk.

A different story exists in the case of the program data, however. This data is accessed at a very high rate and must be in main memory to be effectively accessible by the DATASHARE interpreter. For this reason the program data for all programs is kept resident in main memory. This fact will be shown later to have further advantages in the case of port and printer I/O.

To implement an effective virtual memory accessing algorithm, the program code is kept on the disk as 256 byte pages with one page filling an entire disk sector. Those familiar with DOS will note that this is not compatible with the standard DOS data record format, which allows 253 bytes for user data. The DATASHARE interpreter and compiler have special disk read and write routines to handle this problem. The problem is not as extensive as might be imagined, since only the READ\$ and WRITE\$ routines in the DOS deal with the information in the first three bytes of each data sector. Therefore, all of the space allocation routines in the DOS are still used by DATASHARE. However, none of the standard DOS utilities may be used with the DATASHARE code files. Remember that this concerns only the DATASHARE code files and not the data files.

Because the code is paged in blocks, the DATASHARE programmer can make his program run much more efficiently, in many cases, by forcing his code to cross as few page boundaries as possible. Each time a page boundary is crossed, a new page must be read in. The paging scheme used is purely demand with the least recently used page being destroyed to make space for the new page. Actually, in a lightly loaded system, a single program could get two or three pages all resident in the disk buffer memory at once

and crossing a given page boundary would not cause a disk read, but any significant loading will cause this condition to cease. Therefore, the DATASHARE programmer can assume that each time he crosses a page boundary, a new read will occur. This read can cause from 2 to 130 milliseconds delay in the execution of the program. This time is time that cannot be used by any other program since the disk is busy. By causing an excessive number of page boundary crossings, the programmer can easily cause his program to execute very slowly.

However, an instruction called TABPAGE exists in DATASHARE to aid the programmer in making his execution speed as high as possible. This instruction causes the location counter in the compiler to be incremented until it is at the start of the next page (nothing will be generated if the location counter is already at the start of a page). When this instruction is executed, it causes a GOTO to the start of the next page. By using this instruction, the programmer can cause logical parts of his program to contain as few page boundaries as possible. Another way to increase execution speed is to use in-line coding as much as possible, especially for short operations, instead of the subroutine calling feature if the subroutine is located in a page different from the calling location. This is economically feasible because of the large space available for each program (16K bytes).



## PHYSICAL SYSTEM CHARACTERISTICS

As seen in the map on the previous page, DATASHARE is broken into several major modules. The area between 0 and 05400 contains all of the DOS that is used by DATASHARE. This includes the file loader, basic sector read and write routines (used by the interpreter), and file handling routines.

The area between 05400 and 010000 is used for the interpreter working storage for each port, I/O port buffers, and printer buffer. When a particular user is executed, the 45 bytes corresponding to his interpreter working storage are swapped into the interpreter working storage area. When he stops execution (swapped out for another user to execute), all of this information is swapped back into his area between 05400 and 010000.

The area between 05400 and 010000 is the main working storage page for the entire system. The most actively accessed data is kept within a single page of memory, increasing coding efficiency.

The STATH package used with the DATASHARE 3 system has been revised to require much less space than the package for previous DATASHARE systems. This reduction was effected by grouping functions into subroutines and by taking out the multiplication table. Multiplication now takes approximately the same amount of time as division. The old STATH package was designed for the Version I 2200 and therefore involved a large amount of in-line code which was reduced to subroutines in the new package.

The DATASHARE scheduler is the most complex part of the system. Its task involves all foreground I/O and scheduling of background execution. Background execution is used to interpret and execute the DATABUS statements and perform disk I/O while foreground execution is used to interpret the printer, console, and terminal I/O statements. This portion of the system is explained more thoroughly in the next section.

The DATASHARE interpreter is similar to a standard DATABUS interpreter except that it has been enhanced to deal with based user variable data in the area at the end of main memory and deal with user program data in virtual storage that actually resides on the disk. A base address table exists in the working storage area which tells the interpreter which variable data area to use based on the user number of the user currently being executed. A page address table also exists in the working storage area which tells the interpreter where on the disk the user's program resides. A virtual storage technique is used which uses disk buffers one, two, and three for the storage of the currently active program data pages. When a program data



byte is accessed, the interpreter fetch routine searches through the page address table looking to see if that byte exists in one of the three disk buffers. If the byte does exist, the interpreter merely directly accesses it and the fetch is finished. Otherwise, the interpreter decides which disk buffer has been least recently accessed and reads the necessary program data page into that buffer. The interpreter then goes back and executes the normal fetch routine which will find the byte available in a disk buffer and fetch it for use by the rest of the interpreter.

DATASHARE object code files are structured so that the most significant byte of the DATABUS interpreter program address counter indicates which sector relative to the beginning of the object file and the least significant byte of the address counter indicates which byte within that sector is being accessed. Actually, the first sector of the object file contains the number of bytes that are used for user variable data storage. If this number is greater than the number of bytes of data area allocated to the particular user the program will not be loaded if a CHAIN operation to it is attempted. It turns out that the programmer cannot distinguish between a program actually absent from the DOS directory and a program which has too large a data area to fit into the space allocated to the port trying to load the program.

If the data area will fit in the space allocated to the port, the data area bytes are read into the user's variable data area. Bytes of value 0376 (octal) are not loaded into memory but their slots are skipped. This mechanism allows common variables to be positioned non-destructively. The number of sectors used for variable data storage is kept within a table in the working storage area so the interpreter fetch routine knows by how much to bias the MSB of the program address counter when determining the logical record number of the object code block it needs when obtaining a given program data byte.

### 12.3 Scheduling

To provide optimum response time, DATASHARE handles all port and printer I/O using interrupt driven foreground routines, which means that data transfer between the terminal and the system can occur regardless of the computational task being handled by the background program at any given time. The foreground routines actually interpret the KEYIN, DISPLAY, PRINT, and CONSOLE instructions, with the background interpretive code merely passing these instructions to the foreground through a circular buffer allocated for each port. Conventional systems use such a buffer to hold the actual characters transferred between the system and the terminal. However, DATASHARE uses this buffer to hold the interpretive code bytes, thus enabling many more bytes to be transferred than can actually be held in the buffer. For example, a DISPLAY statement may contain some quoted information and then a variable name. The variable name is represented by two bytes but the contents of the variable could be fifty bytes long, enabling two bytes of buffer space to invoke the transfer of fifty bytes to the terminal. This is made possible by the fact that all program data is resident in main memory which enables the foreground routine to be executing an I/O statement for a given port even though the background program for that port may not be swapped in at the time.

As a matter of fact, the foreground and background program for a given port always execute exclusively of each other to prevent conflicts over data values. When the background program executes a DISPLAY statement, the statement is stored in the buffer for the given port and then the background program is deactivated and the foreground program activated. When the foreground program has completely executed the I/O statement, it causes a high priority interrupt to the background, which deactivates the current program and activates the one which was executing the DISPLAY statement which caused the interrupt. In reality, the scheduling algorithm is more complex than this, but this gives an idea of the sequence of events. One important consideration which must be taken into account by the DATASHARE programmer concerning port I/O is the fact that every time an I/O instruction is completed in the foreground, the background program is swapped in. If the programmer is not careful, he can cause the system to thrash (spend most of its time swapping background programs in instead of doing useful work) by causing a high rate of I/O completion interrupts. An example would be using many separate DISPLAY statements instead of one long continued statement.

The above discussion concerns only port, printer, and

console I/O. All disk I/O is performed under the DOS which is a background-only operation. This means that all DOS functions are non-interruptable and long directory searches (which can take up to several seconds with a four drive system) will cause the response to I/O completion interrupts to be delayed. Long DOS functions, however, occur infrequently and therefore can be ignored from an average response time calculation standpoint.

When the background program resumes execution due to the completion of a foreground I/O task, it is guaranteed a minimum amount of execution time. This prevents the system from spending all of its time swapping background tasks when the foreground I/O completion rate is high.

DATASHARE is capable of driving any serial terminal device which uses an ASCII character set. Use of devices without cursor positioning features, however, will restrict the programmer from using the cursor positioning facility in the KEYIN and DISPLAY statements. If the programmer does not use the cursor positioning feature, he will be able to write a program which is Teletype machine compatible. The \*ES and \*EL list controls send control characters that are ignored by a 35 ASR Teletype. However, the Cursor On character which is sent before each KEYIN variable entry request and the Cursor Off which is sent after the ENTER key is struck, are Tape On and Tape Off respectively on a 35 ASR Teletype.

DATASHARE is also capable of dealing with 103 type datasets as well as hard wired connections and full duplex four wire 202 dataset connections. It handles all of the 103 handshaking involved and needs only the proper cable to work correctly. In fact, the 3360-102 hard wire cable is connected in such a way as to make the 3360-102 appear as a 103 data set, with power on causing ring detect and carrier detect to be sent to the DATASHARE system. The fact that a hard wire or dataset connection is employed at a given terminal cannot be differentiated by the DATASHARE programmer. See Section 13 for more information concerning terminal connections.

## 13. PHYSICAL INSTALLATION

### 13.1 Main peripherals

The DATASHARE system requires a 2200-350 series disk peripheral. Since the system maintains its entire file structure under the DOS, anywhere from one to four disk drives (2.5 to 10 million bytes) may be employed as long as each disk cartridge used has a DOS file structure and the cartridge in drive zero contains the system files. Note that drive zero must be kept on line at all times during system operation but the other three drives may be put on or off line as the maintenance of the data base requires.

Note that, as in any 2200 installation, a 2200-420 parallel interface with an address of 0303 may be connected to drive a special output device, but that device must be capable of handling the output that would normally be given to an ASCII printer.

Besides the 2200-350 series disk, the other required peripheral for the operation of the DATASHARE system is the 2200-460 Multiple Port Communications Interface. In the following discussions the mention of a 2200-460 will imply that a 9900-462 will work as well. As far as DATASHARE is concerned, the 2200-460 and 9900-462 are equivalent. These devices are capable of driving up to eight fully independent full duplex asynchronous lines at speeds ranging from 110 to 9600 baud. The DATASHARE system is not capable of output above 125 characters per second per port and normally uses 1200 baud for direct connection and four wire 202-type modem connections and uses 300 or 110 baud for 103-type modem connections. However, any speed may be strapped in the 2200-460 to achieve compatibility with specific terminals as the occasion may require. The DATAPOINT 3360-102, the recommended terminal device for the DATASHARE 3 system, has switch selectable speeds of 300, 1200, 2400, and 4800 baud. Note that all ports are operated by the DATASHARE system in full duplex mode only.

13.2 Terminal connections

In general, a terminal may be connected to the DATASHARE system in one of three ways: direct hardwire, 103-type modem, and 202-type modem. The following table shows the pin assignments on the 25-pin connector for the 2200-460 individual port, the 3360-102 CRT terminal, and a 103 or 202 type modem:

<u>PIN</u>	<u>2200-460</u>	<u>3360-102</u>	<u>103/202</u>
1	-	PROT GROUND	PROT GROUND
2	DATA OUT	DATA OUT	DATA IN
3	DATA IN	DATA IN	DATA OUT
4	REQ TO SEND	-	REQ TO SEND (202)
5	CLR TO SEND	-	CLR TO SEND
6	-	-	DATA SET READY
7	SIG GROUND	SIG GROUND	SIG GROUND
8	CARRIER DET	-	CARRIER DET
-			
20	DATA TERM RDY	DATA TERM RDY	DATA TERM RDY
-			
22	RING DETECT	-	RING DETECT

The DATASHARE system goes through the following handshaking procedure when a connection is established:

1. Clear Data Terminal Ready and Request To Send
2. Wait for Ring Detection
3. Set Data Terminal Ready and Request To Send
4. Wait up to 10 seconds for Carrier Detect
5. Go to step 1 if time out in step 4
6. Wait one second and then start the ANSWER program

This procedure will work with any of the three types of connections if the proper cable is used.

DIRECT

Basically, the direct connection cable swaps the data wires (pins 2 and 3) and connects Carrier and Ring Detect on one end to Data Terminal Ready on the other as shown in the following table:

2200-460 TO 3360-102 CABLE CONNECTIONS

<u>2200-460</u>	<u>3360-102</u>
2	3
3	2
7	7
8 and 22	20

Note that this arrangement requires only five wires in the cable (four if the optional wire is not used). If the cable is to be made more than several hundred feet long, each of the two signal wires (the ones connecting to pins 2 and 3) should be twisted separately with a ground wire (no other shielding is necessary). Direct connections up to one thousand feet may be made if the above precautions are followed.

The 3360-102 sets Data Terminal Ready whenever it is running. With the above cable connected, this will cause ringing and carrier to be presented to the 2200-460. This has the effect of causing the ANSWER program to be executed whenever power is applied to the 3360-102.

103-TYPE MODEM

The 2200-460 can be connected to a 103-type modem with a one to one cable (e.g., a pin at one end is connected to a pin of the same number at the other end). Only pins 2, 3, 7, 8, 20, and 22 need to be connected but having all pins connected will also work (this being the simplest to describe to someone at a distance!). Note that 103 and 113B modems have similar pin connections.

2200-460 TO 103-TYPE MODEM CONNECTIONS

<u>2200-460</u>	<u>103-TYPE MODEM</u>
2	2
3	3
7	7
8	8
20	20
22	22

If one is calling a 103-type modem over a dial-up network, he will hear the telephone answered very shortly after it starts ringing (should take one or two rings at most). If the telephone is not answered within that amount of time, the caller either has the wrong number or the DATASHARE system is not up or is in the initial phase of being taken down. In any case, the caller may as well hang up (letting the phone ring for a long time can be very irritating at the other end). If the telephone is answered, the caller will hear the carrier from the modem connected to the 2200-460 which is his signal to either depress the DATA key on his modem or put the telephone handset in the data coupler (if he is using one). The DATASHARE system gives the caller ten (10) seconds to perform the necessary action to cause a carrier to be returned from his modem. If all is satisfactorily completed, one more second will pass and then the ANSWER program will begin execution. If all is not

satisfactorily completed, the DATASHARE system will hang up the telephone at its end and go back to waiting for ringing to occur. Note that since the DATASHARE system does wait up to ten seconds for a satisfactory connection, if one dials the system and hangs up as soon as the telephone is answered, he will have to wait ten seconds before he can dial the same telephone again. Also note that the DATASHARE system will disconnect as soon as it loses the Carrier Detect signal from the modem. This means that disconnection will occur even if the carrier is broken only for a very short time.

202-TYPE MODEM

The DATASHARE system requires a full duplex connection to its terminals. A 202-type modem can be used in this fashion only if it is connected via a four-wire circuit. This means that one signal path must exit for data flow in one direction and a separate data path must exit for data flow in the other direction. This implies that a point-to-point connection is made between the modems (the switched telephone network cannot support four-wire connections). In this application, the 202 modem must be strapped for use in four-wire mode.

The connecting cable between the 2200-460 and 202 modem is similar to the one for connection to a 103-type modem except that, since 202's used in point-to-point four-wire service do not use ringing, the carrier detection signal from the 202 must be connected to both the carrier detection and ring detection inputs on the 2200-460.

2200-460 TO 202 MODEM CONNECTIONS

<u>2200-460</u>	<u>202 MODEM</u>
2	2
3	3
4	4
7	7
8 and 22	8
20	20

When Data Terminal Ready is supplied by the terminal device to the remote 202 modem, that modem will turn on its carrier. This carrier will cause the modem connected to the 2200-460 to turn on its carrier detect signal which will present ring detection and carrier detection to the DATASHARE system. The system will proceed to set its Data Terminal Ready signal which will cause the 202 modem to turn on its carrier and complete the connection. One second later the ANSWER program will begin execution. Thus, operation over a 202 modem connection will appear similar to

direct connection operation.

Remote modems are connected to Datapoint 3000 series terminals via a standard modem cable supplied with the terminal. This cable provides the required Data Terminal Ready signal to cause the operational characteristics described above.

### 13.3 Port speed selection

The 2200-460 Multiple Port Communications Adaptor is software programmable to transmit and receive from five to eight information bits with either one or two stop bits. However, the DATASHARE system always uses eight information bits and sends two stop bits (it will receive signals with only one stop bit). The speed of each port may be set independently to a variety of speeds, depending on field programmable hardware straps.

There are three clock buses within the 2200-460, limiting the total number of different speeds used at any one time to three. Each of these buses can be connected to one of two crystal controlled time bases. Each time base is connected to a binary dividing chain, giving speeds selectable in powers of two. The standard crystals supplied provide multiples of 110 and 300 baud. The baud rate of a bus is set by strapping from a baud rate source pin to a baud rate bus input pin. Each bus has eight baud rate output points. The baud rate of a channel is set by strapping from a baud rate bus output point to the channel baud rate input pin. The following table gives the respective pin numbers as found on the silk screening on the printed circuit card in the 2200-460:

BAUD RATE SOURCE		BAUD RATE BUS		
Baud rate	Pin	Bus	Input	Output
300	E29	1	E34	E37
600	E28	2	E35	E38
1200	E27	3	E36	E39
2400	E23			
4800	E22			
9600	E21			
110	E33			
220	E32			
440	E31			
880	E30			
1760	E26			
3520	E25			
7040	E24			
		CHANNEL BAUD RATE INPUT		
		Channel	Input	
		1	E13	
		2	E14	
		3	E15	
		4	E16	
		5	E17	
		6	E18	
		7	E19	
		8	E20	

A typical installation may use baud rates of 110 for teletype machines (remote or local), 300 for remote 3360-102



terminals using 103-type modems, and 1200 for remote 3360-102 terminals using 202-type modems. For this installation, one may connect bus 1 for 110 baud, bus 2 for 300 baud, and bus 3 for 1200 baud as shown in the following table.

E34 to E33	make bus 1 110 baud
E35 to E29	make bus 2 300 baud
E36 to E27	make bus 3 1200 baud

Now, if channels 1 through 3 are to be 300 baud, channels 4 through 7 1200 baud, and channel 8 110 baud, the following connections would be made:

E38 to E13, E14, E15	make ch 1-3 300 baud
E39 to E16, E17, E18, E19	make ch 4-7 1200 baud
E37 to E20	make ch 8 110 baud

Port speeds other than multiples of 110 or 300 baud can be accommodated by changing the crystal frequencies. Selection of the proper crystal should be aided by the Datapoint engineering staff.

#### 13.4 Non-3360-102 terminal devices

Terminals other than the Datapoint 3360-102 can be connected effectively to the DATASHARE system. The major advantage of the 3360-102 is that its cursor can be positioned directly by the issuance of a three character sequence. This allows the usage of the cursor positioning list controls in the DISPLAY and KEYIN statements and greatly enhances the speed of form displays.

Terminals such as the Teletype 33 and 35 KSR or ASR may be connected either hardwire or over modem connections. In addition, conventional CRT terminals such as the Datapoint 3300 (for 300 or 1200 baud) or Datapoint 3000 (for 300 baud only) may be connected. All Datapoint 3000 series terminals use identical cable configurations for a given type of installation. The key to making a cable for a given device is to insure that both Carrier and Ring Detect on the 2200-460 are connected to a wire that is set when the connection is to be established and is cleared when the connection is to be broken.

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 " .

RN	A positive record number ( $\geq 0$ ) 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.
NUL	A null string used as a key to an indexed read.

FOR THE FOLLOWING SUMMARY:

An asterisk \* to the left of an instruction means that the instruction is new to DATASHARE with release 3 or that the instruction format has been substantially enhanced with release 3.

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

*	EQU	10	(a label is required)
*	EQUATE	100	(a label is required)
*	INC	filename[/ext]	
*	INCLUDE	filename[/ext]	

FILE DECLARATIONS

*	FILE
*	IFILE

DATA DEFINITIONS

FORM	n.m
FORM	"456.23"
DIM	n
INIT	"character string"
INIT	"character string"
FORM	*n.m
FORM	*"456.23"
DIM	*n
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	(svar)
*	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)
	LENSSET	(svar)
	CLEAR	(svar)
	EXTEND	(svar)
	LOAD	(svar) FROM (nvar) OF (svar list)
	STORE	(svar) INTO (nvar) OF (svar list)
*	STORE	(slit) INTO (nvar) OF (svar list)
	CLOCK	TIME TO (svar)
	CLOCK	DAY TO (svar)
	CLOCK	YEAR TO (svar)
	TYPE	(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 (nvar list)
	STORE	(nvar) INTO (nvar) OF (nvar list)
*	STORE	(nlit) INTO (nvar) OR (nvar list)

## INPUT/OUTPUT

	KEYIN	(list)
	DISPLAY	(list)
	CONSOLE	(list)
	BEEP	
	PRINT	(list)
	RELEASE	
*	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

### INPUT/OUTPUT LIST CONTROLS

CONTROL	USED IN	FUNCTION
*P<m>:<n>	KDC	Causes the cursor to be positioned horizontally and vertically to the column and line indicated by the numbers <m> (horizontal 1-80) and <n> (vertical 1-24). These numbers may either be literals or numeric variables. Note that <n> is ignored in the CONSOLE statement. This list control is only effective on the Datapoint 3360-102.
*N	KDP	Causes the cursor or printer to be positioned in Column 1 of the next line.
*EL	KDC	Causes the line to be erased from the current cursor position.
*EF	KDC	Causes the screen to be erased from the current cursor position to the end of the line.
*ES	KD	Causes the cursor to be positioned at horizontal position 1 of the top row of the display and the entire display to be erased.
*EOFF	K	Causes the echo during input operations from the terminal to be defeated.
*EON	K	Causes the echo during input operations from the terminal to be
*+	KDCP	Turn on Keyin Continuous for KEYIN or space after logical length suppression for DISPLAY, PRINT, and CONSOLE.
*+	W	Turn on space compression during WRITE.
*-	KDCP	Turn off Keyin Continuous (turned off at the end of the statement) or the space after logical length suppression.

*_	W	Turn off space compression during WRITE.
*<n>	P	Causes a horizontal tab on the printer to the column indicated by the number <n>. No action occurs if the carriage is past the column indicated by <n>.
*<n> *<nvar>	RW	Tab specification for READ or WRITAB operations; the logical file pointers are moved to that character position relative to the current physical record.
;	KDP	Suppress a new line function when occurring at the end of a list.
"	KDCP	Any characters appearing between quotes are displayed or printed when encountered (note that a quote itself cannot be quoted).
*F	P	Causes the printer to be positioned to the top of form.
*L	KDP	Causes a linefeed to be displayed or printed.
*C	KDP	Causes a carriage return to be displayed or printed.
*T	K	Time out after 2 seconds for KEYIN statement.
*W	KD	Pause for one second.

APPENDIX C  
PROGRAM EXAMPLES

Simple ANSWER Program

```
. SIMPLE ANSWER PROGRAM
.
PORTN  FORM "4"
IDCODE DIM 9
ID     INIT "DATAPOINT"
.
      DISPLAY *ES,"D A T A S H A R E   PORT ",PORTN," ON LINE"
      CONSOLE "ANSWER",PORTN
LOOP  KEYIN "ID: ",IDCODE
      MATCH ID TO IDCODE
      GOTO BADID IF NOT EQUAL
      GOTO BADID IF LESS
      MATCH IDCODE TO ID
      GOTO BADID IF LESS
      STOP
BADID DISPLAY "*** INVALID ID ***"
      GOTO LOOP
```



# Simple MASTER Program

. SIMPLE MASTER PROGRAM

.  
PORTN FORM "4"  
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

Complex ANSWER Program

. DATASHARE ANSWER PROGRAM

SYSFILE	FILE		FILE DECLARATION
DAYFILE	FILE		FILE DECLARATION
PORTN	FORM	"3"	THE NUMBER OF THIS PORT
DATE	DIM	18	TODAY'S DATE IN MONTH, DAY, YEAR
IDCODE	DIM	10	
IDCTR	FORM	"3"	
TIMEON	DIM	8	
NFEB	FORM	"29"	
RN	FORM	"000"	
TIME	INIT	"00:00:00"	
DAY	INIT	"000"	
YEAR	INIT	"00"	
NDAY1	FORM	3	
NDAY2	FORM	3	
NYEAR1	FORM	2	
NYEAR2	FORM	2	
LINE	DIM	100	

	DISPLAY	*ES,*N,"D A T A S H A R E	PORT ",PORTN;
	OPEN	SYSFILE,"SYSFILE"	
	CONSOLE	*EL,"ANSWER",PORTN	
STARTO	CLOCK	DAY TO DAY	
	MOVE	DAY TO NDAY1	
	CLOCK	TIME TO TIME	
	CLOCK	YEAR TO YEAR	
	MOVE	NDAY1 TO NDAY1	
	GOTO	NODATE IF ZERO	
	MOVE	YEAR TO NYEAR1	
	MOVE	NYEAR1 TO NYEAR2	
	DIV	"4" INTO NYEAR1	
	MULT	"4" BY NYEAR1	
	COMPARE	NYEAR1 TO NYEAR2	
	GOTO	LEAP IF EQUAL	
LEAP	MOVE	"28" TO NFEB	
	SUB	"31" FROM NDAY1	
	GOTO	JAN IF LESS	
	GOTO	JAN IF EQUAL	
	SUB	NFEB FROM NDAY1	
	GOTO	FEB IF LESS	
	GOTO	FEB IF EQUAL	
	SUB	"31" FROM NDAY1	
	GOTO	MAR IF LESS	
	GOTO	MAR IF EQUAL	
	SUB	"30" FROM NDAY1	
	GOTO	APR IF LESS	
	GOTO	APR IF EQUAL	
	SUB	"31" FROM NDAY1	
	GOTO	MAY IF LESS	

```

GOTO    MAY IF EQUAL
SUB     "30" FROM NDAY1
GOTO    JUN IF LESS
GOTO    JUN IF EQUAL
SUB     "31" FROM NDAY1
GOTO    JUL IF LESS
GOTO    JUL IF EQUAL
SUB     "31" FROM NDAY1
GOTO    AUG IF LESS
GOTO    AUG IF EQUAL
SUB     "30" FROM NDAY1
GOTO    SEP IF LESS
GOTO    SEP IF EQUAL
SUB     "31" FROM NDAY1
GOTO    OCT IF LESS
GOTO    OCT IF EQUAL
SUB     "30" FROM NDAY1
GOTO    NOV IF LESS
GOTO    NOV IF EQUAL
MOVE    "DECEMBER" TO DATE
GOTO    START1

.
NOV     ADD     "30" TO NDAY1
        MOVE    "NOVEMBER" TO DATE
        GOTO    START1

.
OCT     ADD     "31" TO NDAY1
        MOVE    "OCTOBER" TO DATE
        GOTO    START1

.
SEP     ADD     "30" TO NDAY1
        MOVE    "SEPTEMBER" TO DATE
        GOTO    START1

.
AUG     ADD     "31" TO NDAY1
        MOVE    "AUGUST" TO DATE
        GOTO    START1

.
JUL     ADD     "31" TO NDAY1
        MOVE    "JULY" TO DATE
        GOTO    START1

.
JUN     ADD     "30" TO NDAY1
        MOVE    "JUNE" TO DATE
        GOTO    START1

.
MAY     ADD     "31" TO NDAY1
        MOVE    "MAY" TO DATE
        GOTO    START1

.
APR     ADD     "30" TO NDAY1
        MOVE    "APRIL" TO DATE
        GOTO    START1

```

```

MAR      ADD      "31" TO NDAY1
        MOVE      "MARCH" TO DATE
        GOTO      START1

FEB      ADD      NFEB TO NDAY1
        MOVE      "FEBRUARY" TO DATE
        GOTO      START1

JAN      ADD      "31" TO NDAY1
        MOVE      "JANUARY" TO DATE

START1   ENDSET    DATE
        MOVE      NDAY1 TO DAY
        COMPARE   "10" TO NDAY1
        GOTO      START2 IF NOT LESS
        BUMP      DAY
START2   APPEND    DAY TO DATE
        APPEND    ", 19" TO DATE
        APPEND    YEAR TO DATE
        RESET     DATE
        DISPLAY   *+, " ON LINE AT ", TIME, " ON ", DATE
        GOTO      DATEOK

NODATE   DISPLAY   " ON LINE ";
        BEEP
        DISPLAY   " *** DATE NOT INITIALIZED ***"
DATEOK   DISPLAY   " "
        TRAP      LOOP1 IF IO
        OPEN      DAYFILE, "DAYFILE"
        MOVE      "0" TO RN
LOOP0    READ      DAYFILE, RN; LINE
        CMATCH    "9" TO LINE
        GOTO      LOOP1 IF EQUAL
        RESET     LINE TO 72
LOOP0A   BUMP      LINE BY -1
        GOTO      LOOP0B IF EOS
        CMATCH    " " TO LINE
        GOTO      LOOP0A IF EQUAL
LOOP0B   LENSSET   LINE
        RESET     LINE
        DISPLAY   *+, LINE
        ADD       "1" TO RN
        GOTO      LOOP0

LOOP1    KEYIN     *EL, "PLEASE LOG IN: ", *N, IDCODE:
        *C, "*****", *C, "0000000000"
        CLOCK     TIME TO TIMEON
        CONSOLE   *P15:1, *EL, "ID: ", IDCODE, " TIME ON: ", TIMEON
        MOVE      IDCTR TO IDCTR
        GOTO      KABOOM IF ZERO
        MOVE      EIGHT TO RN
LOOP2    READ      SYSFILE, RN; LINE

```

```

      CMATCH " " TO LINE
      GOTO   IDFAIL IF EQUAL
LOOP3 CMATCH IDCODE TO LINE
      GOTO   NEXTID IF NOT EQUAL
      BUMP   LINE
      BUMP   IDCODE
      GOTO   LOOP3 IF NOT EOS
      CMATCH " " TO LINE
      GOTO   NEXTID IF NOT EQUAL
      SUB    "1" FROM PORTN
      WRITE  SYSFILE,PORTN;RN,DATE,TIME
      CLOSE  SYSFILE
      STOP

NEXTID ADD    "4" TO RN
      GOTO   LOOP2

IDFAIL BEEP
      DISPLAY "**** INVALID ID ****"
      SUB    "1" FROM IDCTR
      GOTO   LOOP1

KABOOM CONSOLE *P60:1,*EL,"ID OVERRUN"
      BEEP
      DISPLAY "**** INVALID ID ****"
      GOTO   LOOP1

```

Complex MASTER Program

. DATASHARE MASTER PROGRAM

			FILE DECLARATION
SYSFILE	FILE		THE NUMBER OF THIS PORT
PORTN	FORM	"3"	
ANSWER	INIT	"ANSWERX "	
LINE	DIM	100	
LINITM	DIM	10	
RN	FORM	"000"	
RNX	FORM	"000"	
ONE	FORM	"1"	
FOUR	FORM	"4"	
EIGHT	FORM	"8"	
NINE	FORM	"9"	
TEN	FORM	"10"	
COUNT	FORM	"00"	
CMDLIN	DIM	20	
HELP	INIT	"HELP"	
HELLO	INIT	"HELLO"	
CAT	INIT	"CAT"	
RUN	INIT	"RUN"	
TIME	INIT	"TIME"	
DATE	INIT	"DATE"	
ONLINE	INIT	"ONLINE"	
PORT	INIT	"PORT"	
BYE	INIT	"BYE"	

	RELEASE	
	CONSOLE	"MASTER",PORTN," "
	DISPLAY	*ES
	OPEN	SYSFILE,"SYSFILE"
	SUB	ONE FROM PORTN
	READ	SYSFILE,PORTN;RN
CMDREQ	KEYIN	*ES,*N,"READY",*N,CMDLIN
TRYAGN	MATCH	HELP TO CMDLIN
	GOTO	HELPI IF EQUAL
	MATCH	HELLO TO CMDLIN
	GOTO	HELLOI IF EQUAL
	MATCH	CAT TO CMDLIN
	GOTO	CATI IF EQUAL
	MATCH	PORT TO CMDLIN
	GOTO	PORTI IF EQUAL
	MATCH	TIME TO CMDLIN
	GOTO	TIMEI IF EQUAL
	MATCH	DATE TO CMDLIN
	GOTO	DATEI IF EQUAL
	MATCH	ONLINE TO CMDLIN
	GOTO	ONLI IF EQUAL
	MATCH	BYE TO CMDLIN
	GOTO	BYEI IF EQUAL
	MATCH	RUN TO CMDLIN

```

      GOTO      TRYNAM IF NOT EQUAL
      CALL      GETNAM
TRYNAM  TRAP      CFAIL IF CFAIL
      CLOSE     SYSFILE
      CHAIN     CMDLIN

      OPEN      SYSFILE,"SYSFILE"
      KEYIN     *N,"WHAT?",&N,CMDLIN
      GOTO      TRYAGN

      BUMP      CMDLIN
      RETURN    IF EOS
      CMATCH    "0" TO CMDLIN
      GOTO      GETEXX IF LESS
      CMATCH    ":" TO CMDLIN
      GOTO      GETNAM IF LESS
      CMATCH    "A" TO CMDLIN
      GOTO      GETEXX IF LESS
      CMATCH    "[" TO CMDLIN
      GOTO      GETNAM IF LESS
GETEXX  BUMP      CMDLIN
      RETURN

      KEYIN     *ES,&N:
HELPI    "ENTER: HELLO-<ID> TO SIGN ON AS ANOTHER USER",&N:
      "        HELP      TO GET THIS INFORMATION",&N:
      "        CAT       TO GET A LIST OF PROGRAMS",&N:
      "        TIME      TO GET THE CURRENT TIME",&N:
      "        DATE      TO GET THE DATE AT LOGON",&N:
      "        ONLINE    TO GET THE TIME AT LOGON",&N:
      "        PORT      TO GET THE PORT BEING USED",&N:
      "        RUN-<NAME> TO RUN A PROGRAM",&N:
      "        OR <NAME>  TO RUN A PROGRAM",&N,&N:
      "READY",&N,CMDLIN,*ES
      GOTO      TRYAGN

      CALL      GETNAM
HELLO1  MOVE      CMDLIN TO LINITM
      MOVE      EIGHT TO RNX
HELLO2  READ      SYSFILE,RNX;LINE
      CMATCH    " " TO LINE
      GOTO      IDFAIL IF EQUAL
HELLO3  CMATCH    LINITM TO LINE
      GOTO      NEXTID IF NOT EQUAL
      BUMP      LINE
      BUMP      LINITM
      GOTO      HELLO3 IF NOT EOS
      CMATCH    " " TO LINE
      GOTO      NEXTID IF NOT EQUAL
      READ      SYSFILE,PORTN;RN,LINE
      WRITE     SYSFILE,PORTN;RNX,LINE
      MOVE      RNX TO RN
      GOTO      CMDREQ

```

```

NEXTID  ADD      "4" TO RNX
        GOTO     HELLO2

IDFAIL  BEEP
        KEYIN    "**** INVALID ID ****",*N,"READY",*N,CMDLIN,*ES
        GOTO     TRYAGN

CATI    DISPLAY  *ES,*N,"CATALOG: ",*N
        MOVE     RN TO RNX
        READ     SYSFILE,RNX;LINE
        RESET    LINE TO 11
        MOVE     "9" TO COUNT
        GOTO     CATR1
CATR    READ     SYSFILE,RNX;LINE
        MOVE     TEN TO COUNT
CATR1   RESET    LINITM TO 99
        LENSET   LINITM
        RESET    LINITM
        CMATCH   " " TO LINE
        GOTO     CATR4 IF EQUAL
CATR3   CMOVE    LINE TO LINITM
        BUMP     LINE
        BUMP     LINITM
        GOTO     CATR3 IF NOT EOS
        GOTO     CATRB
CATRA   BUMP     LINITM BY -1
CATRB   CMATCH   LINITM TO " "
        GOTO     CATRA IF EQUAL
        LENSET   LINITM
        RESET    LINITM
        DISPLAY  *+,LINITM
        SUB      ONE FROM COUNT
        GOTO     CATR1 IF NOT ZERO
        ADD      ONE TO RNX
        GOTO     CATR

PORTI   ADD      ONE TO PORTN
        DISPLAY  "YOU ARE ON PORT ",PORTN;
        SUB      ONE FROM PORTN
CATR4   KEYIN    *N,"READY",*N,CMDLIN,*ES
        GOTO     TRYAGN

TIMEI   CLOCK    TIME TO LINE
        DISPLAY  *+,"THE TIME IS ",LINE;
        GOTO     CATR4

DATEI   READ     SYSFILE,PORTN;LINE
        RESET    LINE TO 21
        LENSET   LINE
        RESET    LINE TO 4
        MOVE     LINE TO CMDLIN
        CMATCH   CMDLIN TO " "

```



	GOTO	DATEIN IF EQUAL
	DISPLAY	*+,"THE DATE AT LOG IN WAS ",CMDLIN;
	GOTO	CATR4
DATEIN	DISPLAY	*** DATE NOT INITIALZIED ***";
	GOTO	CATR4
.		
ONLI	READ	SYSFILE,PORTN;LINE
	RESET	LINE TO 29
	LENSET	LINE
	RESET	LINE TO 22
	MOVE	LINE TO CMDLIN
	DISPLAY	*+,"THE TIME AT LOG IN WAS ",CMDLIN;
	GOTO	CATR4
.		
BYEI	CLOCK	TIME TO LINE
	DISPLAY	*+,"LOGGED OFF AT ",LINE
BYEE	KEYIN	CMDLIN
	RESET	ANSWER TO 6
	ADD	ONE TO PORTN
	MOVE	PORTN TO CMDLIN
	SUB	ONE FROM PORTN
	APPEND	CMDLIN TO ANSWER
	RESET	ANSWER
	TRAP	AFAIL IF CFAIL
	CHAIN	ANSWER
AFAIL	GOTO	BYEE

APPENDIX D

FILE ACCESS LOCKOUT PROGRAM EXAMPLE

. FILE ACCESS LOCKOUT EXAMPLE

```

DATAFILE  IFILE
QTYONH    FORM      "0000"
QTYONHS   FORM      "0000"
QTYWD     FORM      "0000"
KEY       DIM       10

          OPEN      DATAFILE,"DATAFILE"

TRYAGN    READ      DATAFILE,KEY;*20,QTYONH;
          MOVE      QTYONH TO QTYONHS
          DISPLAY   "QUANTITY ON HAND: ",QTYONH
          KEYIN     "QUANTITY TO WITHDRAW: ",QTYWD
          SUB       QTYWD FROM QTYONH
          GOTO      ERROR IF LESS
          GOTO      ERROR IF OVER
          PI        5
          READ      DATAFILE,NULL;*20,QTYONH;
          COMPARE   QTYONH TO QTYONHS
          GOTO      TRYAGN IF NOT EQUAL
          SUB       QTYWD FROM QTYONH
          UPDATE    DATAFILE;*20,QTYONH
    
```

APPENDIX E  
COMPILER ERROR CODES

When an E code is given by the compiler at the left of a line of code containing an error, the very next line will contain an asterisk followed by an E code number and another asterisk under the error line at the position of the scanning pointer when the error was detected. The E code number refers to the number in the left column of the following table and the corresponding error explanation in the right column.

- 00001 The first operand of a CMATCH or CMOVE instruction was not an octal number, a quoted character, or a string variable.
- 00002 The second operand of a CMATCH instruction was not an octal number, a quoted character, or a string variable.
- 00003 The second operand of a MATCH or APPEND instruction was not a string variable.
- 00004 The first operand of a MATCH or APPEND instruction was not a string variable or a literal.
- 00005 The first operand of a RESET instruction was not a string variable.
- 00006 The second operand of a RESET instruction was followed by a character that was not a space, implying that there were other operands following the second operand. RESET may have only one or two operands.
- 00007 The first operand of a BUMP instruction was not a string variable.
- 00010 The second operand of a BUMP instruction was not terminated by a space, or had an absolute value of greater than 127.
- 00011 The operand of a CHAIN or ROLLOUT instruction was not a string variable or a literal.
- 00012 The first operand of a STORE instruction was not a string variable or numeric variable or literal. The first operand of a LOAD instruction was not a string

variable or numeric variable.

- 00013 The second operand of a STORE or LOAD instruction was not a numeric variable.
- 00014 The second operand of a STORE or LOAD instruction was not followed by either a space or a comma.
- 00015 One of the third thru Nth operands of a STORE or LOAD instruction was not the same data type as the first operand. If the first operand is a string or numeric variable, then all operands after and including the third operand must be a string or numeric variable, respectively.
- 00016 The second operand of a MOVE instruction was not a string variable or a numeric variable.
- 00017 The second operand of a MOVE instruction was not a string variable or a numeric variable.
- 00020 The first operand of a MOVE instruction was not a string variable or a numeric variable or a literal.
- 00021 The second operand of a COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not a numeric variable.
- 00022 The second operand of a CMATCH, CMOVE, MATCH, APPEND, CHAIN, ROLLOUT, COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not followed by a space (indicating no more operands follow).
- 00023 The first operand of a COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not a numeric variable or a literal.
- 00024 The first operand of an instruction which may be followed by a comma or a preposition was not immediately followed by a comma or a space. If a comma follows the operand a preposition is not looked for. If a space does follow the operand then a preposition must be there.
- 00025 The first operand of a GOTO, CALL, or TRAP instruction was not followed by a space.
- 00026 The first operand of a TRAP instruction was not followed by " IF ".
- 00027 The conditional operand ([NOT] EOS, EQUAL, ZERO, etc.) of a GOTO, CALL, or TRAP instruction was not followed by a space.

- 00030 The conditional operand of a GOTO or CALL instruction was not [NOT] EOS, EQUAL, ZERO, LESS, or OVER; or the conditional operand of a TRAP instruction was not PARITY, RANGE, FORMAT, CFAIL, or IO.
- 00031 The first operand of the TRAPCLR instruction was not followed by a space.
- 00032 The first operand of the TRAPCLR instruction was not PARITY, RANGE, FORMAT, CFAIL, or IO.
- 00033 An operand in a CONSOLE, KEYIN, or DISPLAY instruction was not a string variable or a numeric variable. It was an EQU, FILE, or IFILE variable.
- 00034 A control code (letter or letters following an asterisk) in a CONSOLE, KEYIN, or DISPLAY instruction was not \*C, \*L, \*N, \*T, \*R, \*P, \*EL, \*EF, \*ES, \*W, \*EON, or \*EOFF.
- 00035 A variable <N> in the \*P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was not a number (did not have a first character of 0-9) nor a numeric variable.
- 00036 A variable <N> in the \*P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was a numeric literal with a value for the first (horizontal position) <N> that was not 1 =< <N> =< 80, or with a value for the second (vertical position) <N> that was not 1 =< <N> =< 24.
- 00037 A literal in a CONSOLE, KEYIN, or DISPLAY instruction was not followed by a comma, space, semicolon, or full colon.
- 00040 The last character in the operand string of a CONSOLE, KEYIN, DISPLAY, PRINT, READ, WRITE, or WRITAB instruction was not a space, colon, or semicolon.
- 00041 The end-of-line was encountered before an operand string terminator was encountered for a CONSOLE, KEYIN, DISPLAY, PRINT, READ, WRITE, WRITAB, WEOF, READKS, UPDATE, OPEN, PREPARE, INSERT, or DELETE instruction, or
- The character following the first <N> in the \*P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was not a colon, or
- A quoted string or octal number was specified in the operand string of a READ instruction.

- 00042 An EQUATE, FILE, or IFILE name was specified in the operand list of a PRINT instruction.
- 00043 A character following an asterisk indicating a control code in a PRINT instruction was not +, -, L, F, C, N, or a number 0-9.
- 00044 The first operand of a READ, WRITE, WRITAB, or WEOF instruction was not a FILE or IFILE name.
- 00045 The character following the first operand of a READ, WRITE, WRITAB, or WEOF instruction was not a comma.
- 00046 The second operand of a READ, WRITE, WRITAB, or WEOF instruction having an IFILE name as the first operand was not a string variable name nor a numeric variable name.
- 00047 The second operand of a READ, WRITE, WRITAB, or WEOF instruction having a FILE name as the first operand was not a numeric variable.
- 00050 The character following the first operand of a READKS instruction or the second operand of a READ instruction was not a semicolon.
- 00051 The character following the first operand of an UPDATE instruction or the second operand of a WRITE instruction was not a space or semicolon.
- 00052 An operand in the operand string of a READ or READKS instruction was not a tab (\*<number> or \*<nvar> or \*<EQName>) nor numeric variable nor string variable, or
- An operand in the operand string of a WRITE or UPDATE instruction was not a space compression control (\*+ or \*-) or a quoted string or numeric variable or string variable, or
- An operand in the operand string of a WRITAB or UPDATE instruction was not a tab (\*<number> or \*<EQName>) or space compression control (\*+ or \*-) or quoted string or numeric variable or string variable.
- 00053 A tab operand (\*<number> or \*<EQName> or \*<nvar>) was used in a READ instruction that had an IFILE name as operand one and an NVAR name as operand two.
- 00054 The character following the \* control-indicator character in a WRITE instruction was not a + or -. The compiler will recognize only the \*+ or \*- control

for the WRITE instruction, use the WRITAB instruction to use tab control (\*<number> or \*<nvar> or \*<EQU'd label>) for output to a disk file. For an Index-Sequential file, to use tab control to update a record in the file, use the UPDATE instruction.

- 00055 The operand following an \* control-indicator character was a quoted item. Numeric literals may be used but they may not be enclosed in double-quote " symbols. Numeric literals, numeric variable names, or equated names may be used to specify tab values in KEYIN, DISPLAY, CONSOLE, READ, WRITAB, READKS, or UPDATE instructions.
- 00056 The operand following an \* control-indicator character was not an unquoted numeric literal, a numeric variable name, or an equated name.
- 00057 The first operand of a READKS or UPDATE instruction was not an IFILE name.
- 00060 A tab in a READ, WRITAB, READKS, or UPDATE instruction was greater than 249.
- 00061 A tab in a READ, WRITAB, READKS, or UPDATE instruction was zero. Note that if the value of an EQU'd tab is incorrectly specified the compiler generates a value of zero for the tab, and each use of that tab will generate this error.
- 00062 A character following an operand in the operand string of a READ, WRITE, WRITAB, READKS, or UPDATE instruction was not a space, comma, semicolon, or colon. If the instruction is a WRITAB or UPDATE instruction a semicolon is assumed.
- 00063 The character following the second operand of a WEOF instruction was not a space.
- 00064 The character following the second operand of a WRITAB instruction was not a semicolon.
- 00065 The first operand of an OPEN instruction was not a FILE or IFILE name or the first operand of a PREPARE instruction was not a FILE name.
- 00066 The first operand of a PREPARE instruction was an IFILE name.

There is no provision within the DATASHARE 3 INTERPRETER for the creation of an indexed-sequential file. The file must first exist and be indexed by means of the INDEX program before the file may be

opened by the OPEN instruction and accessed, increased, or decreased by means of the READ, WRITE, WRITAB, WEOF, READKS, UPDATE, and DELETE instructions.

- 00067 The character following the first operand of an OPEN or PREPARE instruction was not a comma.
- 00070 The character following the second operand of an OPEN or PREPARE instruction was not a space.
- 00071 The second operand of an OPEN or PREPARE instruction was not a string variable name or a literal.
- 00072 The end-of-line was encountered before a first operand was encountered in a CLOSE instruction.
- 00073 The first operand of a CLOSE instruction was not a FILE or IFILE name.
- 00074 The character following the operand of a CLOSE instruction was not a space.
- 00075 A character following an operand in a STORE, LOAD, or BRANCH instruction was not a comma, colon, or space.
- 00076 The first operand of a CLOCK instruction was not TIME, DAY, or YEAR.
- 00077 A comma or the preposition TO was not used between the first and second operands of the CLOCK instruction.
- 00100 The second operand of a CLOCK instruction was not a string variable.
- 00101 The character following the second operand of a CLOCK instruction was not a space.
- 00102 The first operand of an INSERT or DELETE instruction was not an IFILE name.
- 00103 The character following the first operand of an INSERT or DELETE instruction was not a comma.
- 00104 The second operand of an INSERT or DELETE instruction was not a string variable name.
- 00105 The character following the second operand of an INSERT or DELETE instruction was not a space.
- 00106 An alphabetic character string where a preposition should have been was not recognized as a preposition:



BY, TO, OF, FROM, or INTO, or

A numeric literal was used but was not enclosed in double quote " symbols.

- 00107 An EQUATE directive was given after an executable instruction was specified.
- 00110 An EQUATE directive was given but no label was specified.
- 00111 The first character of the operand of an EQUATE directive was not 1 thru 9. A first character of 0 implies an octal number which is not allowed in the EQUATE directive.
- 00113 The value specified for an EQUATE directive was not from 1 thru 249.
- 00114 The file specified in an INCLUDE directive was not found on disk.
- 00115 The character after the first operand of a DIM instruction was not a space.
- 00116 The operand value of a DIM instruction was greater than 127.
- 00117 For an INIT instruction or an instruction using a string literal:

No operand was found, or

A character after a quoted string was not comma or space, or

The end-of-line was encountered before the ending quote of a quoted operand was encountered, or

The end-of-line was encountered immediately after a forcing character # was given, or

A character following a comma following a quoted string or an octal number was not a double-quote symbol or a zero, or

A quoted string of greater than 127 characters was specified.

- 00120 For an INIT instruction or an instruction using a string literal:

The character following the ending double-quote

symbol of a quoted string was not a comma or a space.

00121 For an instruction using a string literal: the literal was over 40 characters long.

00122 The end-of-line was encountered before the first operand (data item length specification) was encountered for the DIM instruction.

00123 The end-of-line was encountered before the first operand (numeric data format specification) was encountered, or the numeric data was specified to be more than 22 characters long, for the FORM instruction.

00124 A closing double-quote symbol was not found for the operand (numeric data format specification) of a FORM instruction, or

A numeric literal was used but was not enclosed in double quote " symbols.

00125 For the operand (numeric data format specification) of a FORM instruction or for a numeric literal operand:

The following applies for the FORM instruction if a integer-decimal length was specified:

The character after the first numeric string (specifying the integer part length) was not a space or a decimal point, or

The character after the first numeric string was a decimal point but no numeric string specifying the decimal part length was found.

The following applies if a quoted string was specified:

There were more than 127 characters in the number specification, or

There were no digits specified, or

There was a decimal point specified but no digits followed it, or

The numeric literal was not enclosed in double quote " symbols.

00126 For the DIM, INIT, or FORM instructions: the end-of-line was encountered before an operand was

encountered.

- 00127 An operand was not a quoted item, a number, or a label.
- 00130 The second character after the opening double-quote symbol in the operand of a CMOVE or CMATCH instruction was not a double-quote symbol. The forcing character does not apply in these two instructions because it is not necessary.
- 00131 For an instruction using a literal: the character after the ending double-quote symbol was not a space or comma.
- 00132 An octal number was specified but the number was not in the range 0 thru 037 inclusive.
- 00141 The operand of a PI instruction was not an unquoted numeric literal with a value of 1 through 20.
- 00142 The operand of a WEOF instruction was not an NVAR name.

## APPENDIX F

### INTERPRETER I/O TRAP CODES

- A - an access sequentially by key was attempted before any indexed sequential access was made using the logical file.
- B - the READ mechanism ran off the end of a sector without encountering a physical end of record character (003).
- C - an operation on a closed logical file was attempted.
- D - a non-READ non-DELETE indexed sequential operation was attempted where the specified key already exists in the index.
- E - an EOF mark without at least four zero's was encountered.
- I - the index file specified in an OPEN statement does not exist on the specified drive(s).
- J - the index file found by the OPEN statement does not reside in the correct physical location on the disk (index files may never be moved, they must always be re-created).
- K - a null key was supplied in an operation where the key may not be null.
- M - the data file specified in the OPEN statement does not exist on the specified drive(s).
- N - the data file name specified in the OPEN or PREPARE statement was null.
- O - the index file name specified in the OPEN statement was null.
- P - the file specified in the PREPARE statement had some type of DOS protection (either write, delete, or both).
- T - the tab value in the READ or WRITAB statement was off the end of the sector.
- U - an EOF mark was encountered while a record was being deleted in the indexed sequential file.
- V - one of the indexed sequential access overlays (DS/OV1, DS/OV2, or DS/OV3) could not be loaded by the DOS loader.
- W - an index file pointer sector could not be read.
- X - an index file header sector could not be read.
- Y - the R.I.B. of the data file pointed to by the index file could not be read. (VWXY errors can be caused by parity errors, the drive being switched off line, or the disk cartridge being swapped with another while an operation is taking place.)

