DATASHAR 2.1
~~JANUARY 4, 1973~~
MANUAL UPDATED
SEE FILE 0 7/78

Program Users Guide

Datapoint

DATAPOINT CORPORATION

December 1973

DATASHARE USER'S GUIDE

Release 2

TABLE OF CONTENTS

1. INTRODUCTION

DATASHARE permits the simultaneous execution of up to eight DATABUS programs, each dealing with its own remote Datapoint CRT terminal. The DATASHARE interpreter runs under the Disk Operating System (taking advantage of all of its file handling characteristics), handles a high-speed line printer, 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 assembly and DATABUS language systems may be used alternately to DATASHARE, enabling processing of tasks not applicable to the multiple terminal configuraton.

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 700 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 use of all ports configured into the system. The system may be configured to run with one through eight ports with the data area being divided evenly among them. Thus, an eight port system provides 512 bytes of data area for each program, while a six port system provides 682 and a two port system provides 2048 bytes of data area for each program.

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 system.

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

NOTE: This release of DATASHARE has the following new features which may cause compatibility problems for programs written for the first version of DATASHARE. Refer to the sections indicated for detailed explanations of the features. The pound sign (#) is interpreted by the compiler as a forcing character (Section 2). The OPEN and PREPARE instructions now allow the programmer to specify a drive number (Sections 7.8 and 7.9). The numeric READ no longer uses the MOVE mechanism to reformat the data read in (Section 7.10).

2. STATEMENTS

There are three basic types of statements in DATASHARE: comment, data definition, and program execution. Comment lines begin with a period and may occur 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 understand it more easily. Data definition statements must occur before any program execution statements and are used for setting up all the variables in the program. All data definition statements must have unique labels. Program execution statements must appear after any data definition statements and may or may not have labels. The labels on program execution statements may be the same as labels on the data definition statements. Program execution always begins with the first executable statement. The following are some examples of DATASHARE statements.

```
ONE      FORM "1"
COUNT1   FORM "0"
COUNT2   FORM "0"
PROD     FORM 2
.
START    DISPLAY *ES,"MULTIPLICATION TABLE:",*N
LOOP     MOVE COUNT1 TO PROD
         MULT COUNT2 BY PROD
         DISPLAY COUNT1,"X",COUNT2,"=",PROD," ";
         ADD ONE TO COUNT2
         GOTO LOOP IF NOT OVER
         DISPLAY *N
         ADD ONE TO COUNT1
         GOTO LOOP IF NOT OVER
         STOP
```

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

```
A
ABC
A1BC
B1234
ABCDEF
```

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 emtpy if a space appears in the first column. 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 it 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 ONE TO TOTAL
LABEL2  ADD ONE OF TOTAL      THIS IS A COMMENT
LABEL3  ADD ONE, TOTAL
LABEL4  ADD ONE,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 ONE TOTAL      (missing connective)
LABEL2  ADD ONE ,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.

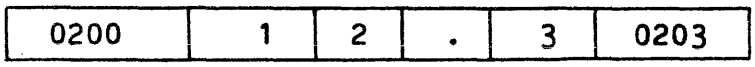
The pound sign (#) is interpreted by the compiler as a forcing character. It may appear in any part of the DATASHARE statement. The character immediately following the pound sign is taken 'as is' regardless of what it is. Thus, the pound sign itself and the quote (") may be used in DATASHARE statements. For example,

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

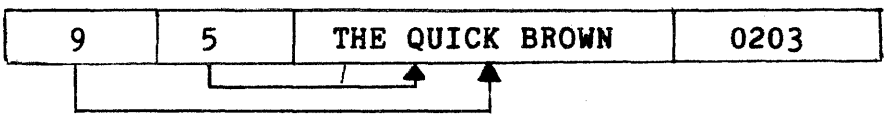
would display exactly, CUSTOMER# SHOULD BE "2222", on the screen. Programs written for the first release of DATASHARE will have to be modified if they use pound signs and are to be compiled by the new compiler.

3. 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:



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. 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:



The first character is called the logical length and points to the last character currently being used in the string (K in the above example). The second character 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.

3.1 Variable definition

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 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).

3.3 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.

3.4 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.

c. 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"
```

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 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:

```
LABEL   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. 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 TSD) is in the specified string, to be loaded and for control to be passed to its first executable statement. Any characters after the sixth will be ignored and blanks will be appended if less than six characters are in the variable. Note that the name used starts at the formpointer. So if in the following example, NXTPGM's formpointer was 4, the chain command would try to load the program named "ROL":

```
NXTPGM  INIT "PAYROL"
        .
        .
        CHAIN NXTPGM
```

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 *
```

appears on the line currently positioned to on the terminal whose program caused the event. The LLLLL is the current value of the program counter and the X is an error letter:

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

Note that the last two items shown above cannot be trapped.

The events that may be trapped are:

PARITY - disk CRC error during READ
disk CRC error during write (write/verify)

RANGE - record number out of range (off end of file, record read which was never written, or WRITAB 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 - the file name supplied was null, a PREPARE was executed using a file that was delete or write protected if it existed, an OPEN was executed that could not find the specified file name, a disk operation was executed using a file that was not open, the drive accessed was off line, space needing to be allocated was not available on the drive accessed, or the number of data characters specified in a WRITAB statement was less than zero or greater than 249.

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. It 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.

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 may be initiated by a DATASHARE program with the following instruction,

```
ROLLOUT (svar)
```

The string variable specifies what function is initially to be executed under DOS. It should be a command line acceptable to the DOS command handler. A CFAIL trap will occur if the string variable is null. For example, the string 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 simple commands,

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

By using the CHAIN command, several DOS functions may 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 at the console.

A DATASHARE program could be written to request from the port the DOS function he wishes to execute and then do a ROLLOUT to that function. A program also could be written to allow a port to create various chain files that might be needed.

The ROLLOUT feature is particularly useful when a file needs to be sorted with the DOS SORT 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.

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.) 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.

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 is of valid numeric format (only digits, spaces, a leading minus sign, and one decimal point allowed). Otherwise, the numeric string is set to zero. 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 would be overstored) are transferred starting with the first physical character in the destination string. The formpointer of the destination string is set to one and the logical length is set to point to 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:

```
STRNG1    4  2    ABCDXLM      ETX

STRNG2    6  3    DOGCAT       ETX

     MOVE STRNG1 TO STRNG2

STRNG2    3  1    BCDCAT       ETX


STRNG1    9  3    AB100.327    ETX

NUMBER    0200    ^39.00       ETX

     MOVE STRNG1 TO NUMBER

NUMBER    0200    100.33       ETX


NUMBER    0200    100.33       ETX

STRNG1    9  3    AB100.327    ETX

     MOVE NUMBER TO STRNG1

STRNG1    6  1    100.33327    ETX
```

5.2 APPEND

APPEND appends the source string to the destination string. 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:

```
STRNG1     8  6     JOHN^DOE    ETX

STRNG2    11  11    MARY^JONES^^^^^^^^^^    ETX

      APPEND STRNG1 TO STRNG2

STRNG2    14  14    MARY^JONES^DOE^^^^^^^   ETX
```

5.3 MATCH

MATCH compares two character strings starting at the formpointer of each and stopping when the end of either string is reached. If either formpointer is zero, 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 numeric value of the destination character is less than the numeric 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:

SOURCE	DESTINATION	RESULT
ABCDE	ABCD	EQUAL,LESS
ABC	Z	NOT EQUAL,NOT LESS
ZZZ	AAA	LESS,NOT EQUAL
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, 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 alphanumerics, 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 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 32 (space gives zero, ! one, " two, 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 32 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.

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 formponter 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

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

Example:

```
LENSET 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).

Example:

```
STORE Y INTO NUM OF ITEM,ENTRY,ALINK
```

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

Examples:

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

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 not incremented. This implies that, if the DATASHARE system is running 24 hours a day and is using the date, it will have to be taken down at midnight to reset the clock. 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.

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 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 indicates that the content of the second variable is negative following the execution of the instruction (or would have been in the case of COMPARE). ZERO (or EQUAL) indicates that the value of the second variable is zero following the execution of the instruction.

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 a variable was defined for, the result is rounded up. 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
```

Note that if an OVER occurs during an ADD, SUB, or COMPARE of two strings of different physical lengths, the result 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
```

6.2 SUB

SUB causes the content of variable one to be subtracted from the content of variable two.

Examples:

```
SUB RX350 FROM TOTAL
SUB Z,TOTAL
```

6.3 MULT

MULT causes the content of variable two to be multiplied by the content of variable one.

Examples:

```
MULT B BY A
MULT W,Z
```

6.4 DIV

DIV causes the content of variable two to be divided by the content of variable one. The number of decimal places in the result is equal to the number of decimal places in variable two minus the number of decimal places in variable one. If this number is negative, it is assumed to be zero.

Examples:

```
DIV SFACT INTO XRSLT
DIV 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 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 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
```

7. INPUT/OUTPUT INSTRUCTIONS

The DATASHARE statements that actually move data between the program variables and the terminal, printer, or disk, all 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 increases execution speed greatly.

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 to 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 stored

into. 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.

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. 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, and the *N control causes the cursor to be set to the first column of the next line.

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:5,"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). Normally, when the cursor is not flashing, all characters will be ignored. The exception, however, is the interrupt character, which may be keyed at any time and will result in a 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). 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 formpointers 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.

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 displayed instead of keyed in on 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 nothing is 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.

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. 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. All other DISPLAY list controls, except for the *C, are operative. 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 to 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 one 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 from each terminal and the name of the program to which a CHAIN was last executed as well as the current time.

Example:

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

7.4 BEEP

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

7.5 PRINT

DATASHARE supports one local printer. The printer may be accessed on a sequential 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;
```

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 interrupt character, the printer is automatically released.

7.7 PREPARE

PREPARE creates a new DOS file with the name given in the string variable specified. 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 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. This is a new feature included in this version of DATASHARE and may effect programs written for the previous DATASHARE version. Programs should be checked to be sure that their file name variables will not assign drive numbers unintentionally when used under the new DATASHARE version.

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 I/O error will occur. The logical record number limit is always set to 9696 by the PREPARE instruction.

One always deals with "logical files" in DATASHARE once he has opened them with either the PREPARE or OPEN instructions. A terminal may have up to three logical files (numbered 1, 2, and 3) which are specified by a logical file number in all disk I/O instructions.

For example, let the following strings be defined as follows:

```
FNAME   INIT "SOURCE"
GNAME   INIT "SCRATCHX1"
HNAME   INIT "F1FILEOF1FILE1"
```

Let the formpointer of FNAME be 1 and its logical length be 6, let the formpointer of GNAME be 1 and its logical length be 9, and let the formpointer of HNAME be 8 and its logical length be 13. If the following PREPARE operations were executed:

```
PREPARE 1,FNAME
PREPARE 2,GNAME
PREPARE 3,HNAME
```

the file SOURCE/TXT would be prepared as logical file 1 on the first drive (beginning with drive 0) on which space was available, the file SCRATCHX/TXT would be prepared as logical file 2 on drive 1 (if no space was available or the drive was off line, an I/O error would occur), and the file F1FILE/TXT would be prepared as logical file 3 on drive 1.

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.

7.8 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 I/O 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.

For example, if the following operations were performed:

```
FILE1   INIT "F1NAME"
FILE2   INIT "DATAFILE3"
        .
        .
        OPEN 3,FILE2
        OPEN 2,FILE1
```

all drives beginning with drive 0 would be searched for file F1NAME/TXT. Only drive 3 would be searched for file DATAFILE/TXT. An I/O error would occur if drive 3 were off line.

If the user plans to deal with a very large file in a random access fashion, 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 9000 records (the exact limit depends upon the amount of program information kept on the particular disk).

7.9 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 3
```

7.10 DISK INPUT/OUTPUT

DATASHARE disk files may be random or sequential. A random file would have a well defined correspondence between logical and physical records with each physical record containing up to 249 data characters:

(data)(015)(003)

A sequential file may have any number of logical records per physical record with some logical records crossing physical record boundaries. The following shows five logical records contained within three physical records:

```
Physical record n:    (data)(015)(data)(015)(003)
Physical record n+1:  (data)(015)(data)(015)(data)(003)
Physical record n+2:  (data)(015)(003)
```

When accessing a file, a record number is always specified. A random access is indicated by a record number greater than or equal to zero corresponding to the physical record desired. A sequential access is indicated by a record number less than zero. The actual record number specified for a sequential access has no significance. The interpreter only looks to see if the variable is negative. In the following discussions and examples, RN will denote a positive record number and SEQ will denote a negative record number (e.g. -1).

When a read or write statement is executed, there are two conditions to consider: the position in the file where reading or writing begins and the position in the file where reading or writing stops. The difference in the possible write and read statements is the difference in these two conditions. A write or read operation always begins at the current position of the logical file. The current position of each logical file open is defined by two pointers: a physical record pointer and a character pointer. The physical record pointer refers to the current physical record of the logical file. This pointer corresponds to the DOS logical record number. The character pointer refers to the current character position (1 to 249) within the current physical record. When a file is opened, its physical record pointer is initialized to zero and its character pointer to one. The pointers are then moved with the various types of read and write operations.

Space compression may be used when writing files. It is most useful for sequential files but may be used on random files. When space compression is on and two or more consecutive spaces are to be written, two octal bytes are written instead of the full number of spaces. The first byte is the octal number 011. This byte indicates that

space compression count follows. The second byte is the number of spaces that have been compressed. When the space compression characters are read, the spaces are expanded to their full number. A 011 is never allowed to be written as the 249th character of a physical record. If this case should occur, the physical record is terminated and both space compression characters are written in the next physical record.

Space compression is turned on by the list control, *+, in a write statement. It is also turned on at the execution of any read statement. When space compression is first turned on, the space compression counter is set to zero. Space compression is turned off at the start of a random WRITE, WRITAB, or WEOF. No change occurs in the space compression status with the execution of a sequential WRITE, WRITAB, or WEOF.

Note that sequential files may contain logical records of extreme length through the use of continued writing. One must be careful in the use of logical records longer than 255 characters when using space compression because the count may be overflowed. If an attempt is made to compress more than 255 spaces, the actual number written will be the number attempted modulo 256.

Also note that when space compression is on, trailing spaces in a logical record are not written since the last space compression count is simply dropped and the 015 is written to signify the end of the logical record. This action has consequence when space compressed records are read (see the section on the READ statement).

RANDOM or SEQUENTIAL WRITE

The write statement consists of a logical file number and a record number followed by a list. The list may include variable names, quoted characters, list controls, and octal control characters (000 through 037). Each character string variable 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, form pointer, 0200, or ETX). The quoted items and octal control characters will be written exactly as they appear in the list. The list controls are used for write tabbing and space compression. For example,

```
WRITE 3,RN;"TIME: ",TIME,015,"TOTAL: ",TOTAL
```

The following is a list of the different types of write statements. Remember that SEQ is a numeric variable with a negative value and RN represents a numeric variable with some specific non-negative value.

(1) WRITE 1,RN;A,B,C

This is a normal random write. The physical record pointer would be set to RN and the character pointer to one. Variables A, B, and C would then be written followed by an end of logical record (015) and end of physical record (003). The character pointer remains pointing at the 003.

(2) WRITE 1,RN;A,B,C;

This is a random write to be continued as indicated by the semicolon at the end of the statement list. The physical record pointer would be set to RN and the character pointer to one. Variables A, B, and C would then be written without any trailing characters (no end of logical record or physical record). The character pointer would be left pointing one character position past the last character written for variable C.

(3) WRITE 1,SEQ;A,B,C

This is a normal sequential 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 character position one in record zero of logical file one. Otherwise, it would be positioned according to the results of the last write or read statement executed. An end of logical record (015) and end of physical record (003) are written after the last character in variable C. The character pointer is left pointing at the (003).

(4) WRITE 1,SEQ;A,B,C;

This is a sequential write to be continued as indicated by the semicolon at the end of the statement list. Variables A, B, and C are written beginning at the character position currently being pointed to by the logical file pointers. No other characters are written and the character pointer is left pointing one character position past the last character written for variable C.

(5) WRITAB 1,RN;A,*70,B,*10,C,*NVAR,"TIME"

This is the write tab feature which requires a different instruction mnemonic. 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 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 that point.

Tab positioning in random write tabs is calculated from the first position of the physical record specified. 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 position past the last character written.

The above example would write variable A beginning at character position one of physical record RN, variable B beginning at character position 70, variable C beginning at character position 10, and the characters "TIME" beginning at the character position indicated by numeric variable NVAR. 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.

The write tab may also be used to position to a particular place in a file when it is desired that space compression be left off. For example,

WRITAB 1,ZERO;*1

would position to the beginning of logical file one. If space compression need not be left off, file positioning can be performed more efficiently with the READ operation.

Note that using write tab with a negative record number is possible but is not advisable. Tab positioning would be calculated from the first character position in the current physical record. Strange results may occur if the programmer is not aware of this fact. In order to use the write tab on sequential files, he would have to know exactly where

the logical records were placed within the physical records.

To summarize disk write operations, a random write starts with character position one of the specified physical record number except when specified otherwise in a write tab operation in which case the position may be specified with a list control. Sequential writes start with the current character position pointed to by the logical file pointers. For all write statements except the write tab, if the statement list is not terminated by a semicolon, an end of logical record (015) and end of physical record (003) are written with the character pointer left pointing at the 003. If a semicolon terminates the statement list or the statement is a write tab, no end of logical record (015) or end of physical record (003) will be written and the character pointer is left pointing one position past the last character written.

Thus, if example (1) or (3) were followed by a sequential write, the 003 would be overstored by the first data character of the new logical record written. If example (2) or (4) were followed by a sequential write, the data written would be a continuation of the same logical record.

WRITE END OF FILE

Standard DOS end of file marks (000 000 000 000 000 000 003 in the first 7 character positions of a physical record) may be written in DATASHARE. WEOF does not change the physical record or character pointers for the file.

(6) WEOF 1,RN

This statement writes an end of file mark in physical record RN.

(7) WEOF 1,SEQ

This statement would cause an end of file mark to be placed in the next physical record into which the first seven data characters may be written.

RANDOM or SEQUENTIAL READ

The read statement consists of a logical file number and a record number followed by a list. The list may include variable names or a list control used for read tabbing. When a numeric item is read, the number of characters corresponding to the length of the variable are

read in. Any non-leading blanks read would be converted to zeros (e.g. ^3^2^1 would be read as ^30201). If a non-numeric character other than a negative sign as the first non-blank character, decimal point or blank 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.

Note that the numeric read in DATASHARE 2 is different from that in DATASHARE 1 (which read the number into a temporary location and then used the MOVE mechanism to tranfer the data to the actual numeric variable, allowing for reformatting and rounding). This condition may effect programs written for DATASHARE 1 and should be considered if chaining to the DATASHARE 2 system.

When a string is read, the number of characters corresponding to the 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 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.

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.

The following is a list of the different types of read statements:

(8) READ 1,RN;A,B,C

This is a normal random read. The physical record pointer is set to RN and the character pointer to one. Variables A, B, and C are then read. Any remaining characters in that physical record are discarded since the character pointer is left pointing at the 003 for that physical record.

(9) READ 1,RN;A,B,C;

This is a random read to be continued as indicated by the semicolon at the end of the statement list. The physical record pointer is set to RN and the character pointer to one. Variables A, B, and C are then read. The character pointer is left pointing one character position past the last character read for variable C.

(10) READ 1,SEQ;A,B,C

This is a normal sequential read. Variables A, B, and C are read from logical file one beginning at the current character position pointed to by the logical file pointers. Any characters left over in that logical record are discarded since the character pointer is left pointing one character position past the end of that logical record (015).

(11) READ 1,SEQ;A,B,C;

This is a sequential read to be continued as indicated by the semicolon at the end of the statement list. Variables A, B, and C are read beginning at the current character position pointed to by the logical file pointers. When reading is completed, the character pointer is left pointing one character position past the last character read for variable C.

(12) READ 1,RN;A,*100,B,*NVAR,C,*50,D

By including the list controls in the read statement above, specific characters may be read from a

record. The list controls *(numeric literal) or *(numeric variable) are used to position the pointer to the specified character position in the specified physical record. Reading of the variable begins at that point.

Tab positioning in random read operations is calculated from the first 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 end of record mark following the last character read if the statement list is not terminated by a semicolon. If it is terminated by a semicolon, the character pointer is left pointing one character position past the last character read.

The above example would set the physical record pointer to RN and the character pointer to one. Variable A would then be read. Variable B would be read beginning at character position 100, variable C beginning at the character position indicated by the numeric variable NVAR, and variable D beginning at character position 50. The character pointer would be left pointing at the 003 following the last character read into variable D.

Read tabbing may be used to position to a particular place in a file. For example,

```
READ 1,ZERO;*1;
```

would position to the beginning of logical file one. Remember, however, that if this positioning is performed for writing purposes, space compresssion will be on for the write operation.

Note that using the read tab with a negative record number is possible but not advisable. Tab positioning is always calculated from the first character position in the current physical record. Strange results may occur if the programmer is not aware of this fact.

To summarize disk read operations, a random read starts with character position one of the specified physical record number. If tab list controls are included, the variables are read beginning at the specified character position relative to the beginning of the current physical record. Sequential read operations begin with the current character position pointed to by the logical file pointers. If the statement is random and is not terminated by a semicolon, the character pointer is left pointing at the end of record

mark following the last character read. If the statement is sequential and not terminated by a semicolon, the character pointer is left pointing one character position past the end of that logical record (015). For all read operations, if the list is terminated by a semicolon, the character pointer is left pointing one character position past the last character read.

Thus, if statement (8) was followed by a sequential read, the logical record in the next physical record would be read. If statements (9) or (11) were followed by a sequential read, the rest of the characters in that same logical record would be read. If statement (10) was followed by a sequential read, the next logical record in the file would be read.

TEST FOR END OF FILE

A test of the OVER flag may be made to determine if an end of file was read. The test should be made after the read statement. For example,

```
READ 1,RN;A,B,C
GOTO LABEL IF OVER
```

If an end of file is read, the variables in the statement will be set to zero or have their logical lengths and formpointers set to zero depending on whether they are numbers or strings. Note that the OVER flag is not set if a READ to be continued is executed (semicolon at the end of the statement list).

8. PROGRAM GENERATION

8.1 Preparing Source Files

Files containing the source language for DATASHARE programs are prepared using the general purpose editor running under the DOS. These files are prepared identically to preparing DATABUS source files since DATASHARE runs programs written in the DATABUS language. The use of the general purpose editor running under the DOS is covered in a separate document. This editor has a DATABUS mode providing a tab stop to make the text more readable.

8.2 Compiling Source Files

DATASHARE programs are compiled using the DATASHARE compiler running under the DOS. Note that DATASHARE programs must always be compiled using the DATASHARE compiler running alone under the DOS. This implies that programs cannot be generated while the DATASHARE system itself is executing (the DATASHARE system must be stopped either manually or with the use of ROLLOUT). The DATASHARE compiler is parameterized in the following manner:

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

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 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 will be searched beginning with drive 0 for the source file. The first thing the compiler does is try to find the source file.

The object file will contain the object code generated by the compiler from the specified source code. If it is not given, the name of the source file with an extension of TSD is assumed. Note that DATASHARE can run only those files with extension TSD. If the file is specified without a drive number, it will be placed on the same drive as the source file.

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 program when the DATASHARE system was restored. This procedure would shorten the total time that the 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 used under DATASHARE, it must have an extension of TXT. If no drive number is specified, the print file will be placed on the same drive as the source file.

Output Parameters

These parameters allow the user to specify what type of output he wants 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 local printer. If the semicolon but no parameters are specified, the only output is the object file (if a print file was specified, it would be null). If no semicolon is typed, the compiler asks the operator the options step by step. Any lines which have

errors are always displayed on the screen with the appropriate error flag.

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. 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 from the bottom of the current page.

X A cross-reference listing is printed at the end of the compilation for the data variables defined and for the statement labels defined. Each cross-reference is sorted alphabetically. When the cross-reference is printed, the data variable name or label symbol is given and preceded by the octal location where the item was defined. Following the name is a list of all line numbers in which the item was defined or referenced. An asterisk flags those line numbers which are definitions. If a cross-reference is requested, the following messages will be displayed as the data variables and labels are being sorted and printed:

```
SORTING DICTIONARY
FINAL MERGE/PRINT -- DATA XREF
FINAL MERGE/PRINT -- XEQT XREF
```

A cross-reference may be obtained regardless of whether a listing was requested.

D A copy of the source and object code is displayed on the screen during the compilation.

C If a listing was requested, the output will normally consist of the source code preceded by the starting octal location for that line. The actual object bytes generated will also be printed if the user specifies this parameter. Printing the object code usually makes the listing about twice as long. If this option is given, the L option is implied and therefore need not also be supplied.

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 also specified since those listings will automatically include error flags.

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/TSD. 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/TSD. A listing would be printed on the local printer and consist of the source and object code with a data and 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/TSD 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 is turned on and execution is not resumed until the DISPLAY key is depressed again (the DISPLAY light is turned off). Compilation may be aborted at any time before the cross-reference sort is begun by depressing the KEYBOARD key.

8.3 Compilation diagnostics

The compiler prints and displays diagnostic messages on the listing to help the programmer debug syntatical errors in his code. These messages take the form of an error code letter at the left 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 syntatical error), U for an undefined variable or label, and I for an undefined instruction. 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 in a fashion similar to the way it flags the file if syntax errors occur. In this case, the object file for the given program should be deleted from the DOS (using the KILL command) and the program re-compiled (without a listing). Segment errors usually occur when a program has been edited such that its object file becomes longer and will no longer fit in the segment previously allocated to it. If the object file is very long and the disk extremely fragmented then deleting the file and re-compiling may not solve the problem. In this case, the disk should be purged or the BACKUP program used (if a dual drive system is available) to make more contiguous free space available.

9. SYSTEM GENERATION

9.1 Loading From Cassette

The DATASHARE compiler and interpreter system programs are contained on one cassette.

The compiler files cataloged on the cassette are CMPCMD, CMPOV1, and CMPOV2. They should be cataloged on the disk using the DOS commands:

```
IN DSCMP/CMD,N5
IN DSCMP/OV1,N6
IN DSCMP/OV2,N7
```

The interpreter files cataloged on the cassette are DS, DSOV1, DSCON, DSIN, DSOUT, DSBACK, ROLOUT, IHAND, and ROLFIL. The files should be cataloged on disk with the DOS commands:

```
IN DS/CMD,N2
IN DS/OV1,N3
IN DSCON/CMD,N4
IN DSIN/CMD,N10
IN DSOUT/CMD,N11
IN DSBACK/CMD,N12
IN ROLLOUT/SYS,N13
IN INTRHAND/SYS,N14
IN ROLLFILE/SYS,N15
```

The first three files are necessary for the DATASHARE system to run without the ROLLOUT feature. Files 7 through 12 are necessary if the ROLLOUT feature is to be used (see Section 4.9).

DSCON/CMD is a program to configure the system for a given number of ports.

DSIN/CMD and DSOUT/CMD are substitute programs for the normal DOS commands IN and OUT. Since DATASHARE object files have a special disk format that is not compatible with normal DOS file format, the DOS utility commands cannot be used with DATASHARE object files. DSIN/CMD and DSOUT/CMD must be used instead.

9.2 Port Configuration

The DATASHARE system may be configured to run with from one to eight ports. The total data space for the DATASHARE system is 4096 bytes and is divided evenly among the ports configured for the system. Therefore, systems requiring fewer ports can have more data space allocated for each one.

The system is configured by running the DSCON program. This program will ask:

NUMBER OF DATASHARE PORTS?

to which the response should be a digit between one and eight. If one of these digits is not given in response, an error message will be given and the request repeated. Once a valid response has been given, control will be returned to the DOS.

Note that the number of ports configured for the system may be changed at any time. The compiler generates code independently of this number, allowing the user to create a data space as large as necessary. However, when one chains to a program it will appear not to exist if its data space will not fit within the limits incurred by the current number of ports configured (4096 divided by the number of ports).

The number-of-ports information is kept in logical record number three of the DSCON/CMD object file to eliminate the need for a separate file. This can be done since the DSCON/CMD program is very short. The DS/CMD program accesses the DSCON/CMD file by that name, thus the requirement that the name be DSCON/CMD when the system is brought up.

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.

10. SYSTEM OPERATION

10.1 Bringing Up the System

The DATASHARE system is brought up by entering the DOS command:

DS

This runs a very short program which loads the main system file, DS/OV1, into memory. If this file cannot be loaded, the message:

*** DS/OV1 MISSING ***

is displayed and the machine beeps and halts. Otherwise, the port configuration information is read. If the file DSCON/CMD cannot be found, the message:

* DSCON/CMD MISSING - 8 PORTS ASSUMED *

is displayed. If the file can be found but the port number information is not correctly formatted (this will occur if the DSCON program has never been executed), the message:

* DSCON/CMD BAD - 8 PORTS ASSUMED *

is displayed. The system then 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. The DS/CMD program must be AUTO'ed under the DOS and the auto-restart tab on the DOS boot tape punched to enable the DATASHARE system to restart after a power failure. 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 15 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 15 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.

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 which should close all three logical files. 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 envoke 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 overlayed 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. If the auto-restart tab is punched on a DOS bootstrap tape in the rear deck, the halting will cause the DOS to be fully restored.

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.

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 for a given port, the port will simply not be activated when the system is brought up.

Note that the first thing any of the examples do is execute a CLOSE statement for each file. The ANSWER program should do this to properly close all files whenever a port disconnects. The MASTER program should do this to properly close all files whenever a STOP, chain failure, or INTERRUPT key occurs. The simple ANSWER program then 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 closes all files (in case the INTERRUPT key was struck) and then 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 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 either with DATABUS 7 or DATASHARE, the latter 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.

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 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 of 256, the DATASHARE programmer can make his program run much more effeciently, 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

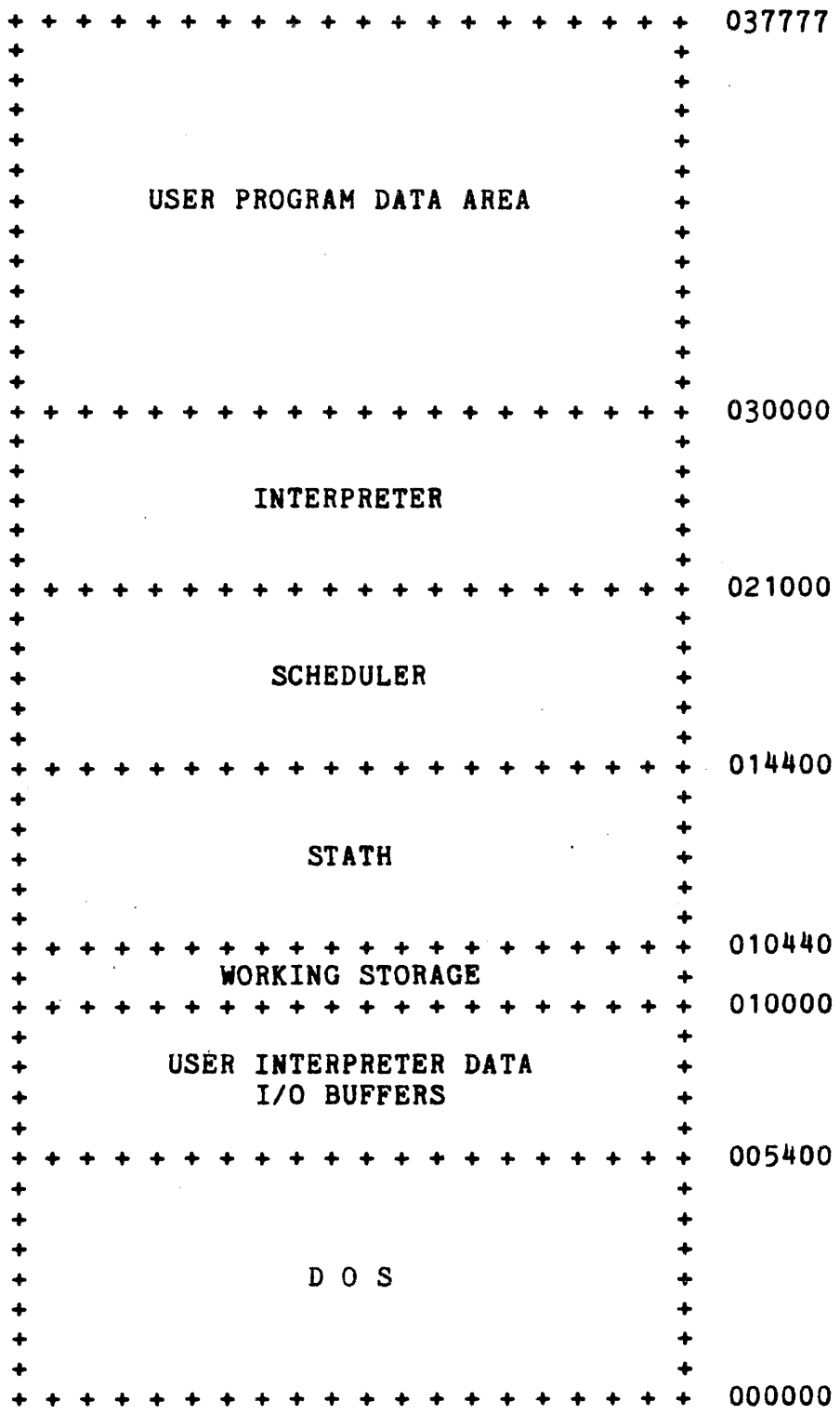
once and crossing a given page boundary would not cause a disk read, but any significant loading will cause this condition to cease. Therefore, the DATASHARE programmer can assume that each time he crosses a page boundary, a new read will occur. This read can cause from 2 to 130 milliseconds delay in the execution of his 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).

Since all program data is resident, the amount of space available to each port is limited. A total of 4096 bytes of space is allocated for the combined use of all ports. For an eight port system, this amounts to 512 bytes of data space per port. However, the system is configurable to allow fewer ports to be used in a system and correspondingly more data area per port. The data area is always evenly proportioned among all of the ports configured into the system, and therefore is equal to the greatest interger value of 4096 divided by the number of ports. This number is 512, 585, 682, 819, 1024, 1365, 2048, and 4096 for 8, 7, 6, 5, 4, 3, 2, and 1 port systems respectively.

12.2 Major Modules

Memory map of the DATASHARE run-time system:



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 user logical files tables, interpreter working storage, I/O port buffers, and printer buffer. When a particular user is executed, the 48 bytes corresponding to his three available logical files are swapped into the DOS logical file table and the 43 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 010000 and 010440 is the main working storage for the entire system. The most actively accessed data is kept within a single page of memory, increasing coding effeciency.

The STATH package used with the DATASHARE system has been reduced in size by removing the keyin and display routines. In all other respects, it is similar to the package used in the other DATABUS interpreters.

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 stored with 256 bytes per disk sector. This enables the most significant byte of the DATABUS interpreter program address counter to indicate which sector relative to the beginning of the object file and the least significant byte of the address counter to indicate which byte within that sector is being accessed. Actually, the first sector of the object file contains the number of sectors that were used for user variable data storage. These sectors are read into the user's variable data area when a chain is made to the program. Bytes within these sectors set to 0377 (octal) are not loaded into memory but their slots are skipped. This mechainism allows common variables to be positioned non-destructively. If, while loading the data area, the interpreter exhausts the data space allocated to a single user, a chain failure is initiated. Therefore, the programmer cannot distinguish between a program actually absent from the DOS directory and one whose data area will not fit into the space allocated to an individual program for the number of ports currently configured. 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. The minimum execution time can be used to advantage when one is dealing with common data base file access. It is only structurally sound to allow one port to modify a given record in a file at any given time. For example, picture an inventory file. Let us assume that a quantity of ten exists at some point in time for a given item. Let us also assume that two ports are simultaneously attempting to deplete this quantity by one. It is apparent that it would not be very difficult for the first port to read the file and then be interrupted by the second port which proceeds to read the file, deplete the quantity by one, and modify the file with the new value. Then the first port resumes execution and depletes the quantity which it had originally read and modifies the file with its new value. Unfortunately, this new value is wrong since the quantity read by the first port was out of date when its execution resumed. For this reason, the first port should have locked out access to the given record (or even file) while it was trying to change its contents.

Access could have been locked out if the first port knew that it could have read, depleted, and rewritten the quantity without being interrupted by the second port. If before doing the file access, the first port had just completed a foreground I/O operation (for example, a DISPLAY statement), a minimum execution time of 800 milliseconds would have been assured. Since a disk access consumes a maximum of 180 milliseconds, this would be an adequate amount of time to read the record, perform a simple computation, and rewrite the record. This is the recommended technique of common file access lock out. If a foreground I/O statement has not just been completed, the program can perform the statement:

```
CONSOLE *P1:79
```

which requires very little execution time and performs no visible operation. Execution of this statement will cause the user to be swapped out and back and may cause some delay while another program executes, but must be performed if the

800 milliseconds of execution time is to be assured. Single character string operations occur within five milliseconds, multiple character string operations occur within five to ten milliseconds, depending upon the length of the strings, and arithmetic operations occur within five to fifty milliseconds depending upon the operation being performed (addition and subtraction being the shortest and division the longest) and the length of the numbers.

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.

Any 2200-200 series printer which uses the ASCII character set and requires no special motion controls may be used as the local printer on the DATASHARE system. Note that the current release of the software excludes the use of the 2200-250 series servo printer. Only one printer may be connected and must have the I/O bus address of 0303. Note that, as in any 2200 installation, a 2200-420 parallel interface 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. This device is 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 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:

PIN	2200-460	3360-102	103/202
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

2200-460	3360-102
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

2200-460	103-TYPE MODEM
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

2200-460	202 MODEM
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 hardwire 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	Pin
300	E29
600	E28
1200	E27
2400	E23
4800	E22
9600	E21
110	E33
220	E32
440	E31
880	E30
1760	E26
3520	E25
7040	E24

BAUD RATE BUS

Bus	Input	Output
1	E34	E37
2	E35	E38
3	E36	E39

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 except for a quote (").
event	The occurrence of a program trap: PARITY, RANGE, FORMAT, or CFAIL.
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 six characters are used).
label	A name assigned to a statement.
nvar	A name assigned to a directive defining a numeric string variable.
nval	A name assigned to a directive defining a numeric string variable or an immediate numeric value.
nlit	An immediate numeric value.
svar	A name assigned to a directive defining a character string variable.
sval	A name assigned to a directive defining a character string variable or a quoted alphanumeric character.
RN	A positive record number (>= 0) used to randomly READ or WRITE on a file.

SEQ	A negative number (< 0) used to READ or WRITE on a file sequentially.

DIRECTIVES

```
FORM n.m
FORM "456.23"
DIM n
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)
TRAP (label) IF (event)
TRAPCLR (event)
ROLLOUT (svar)
```

CHARACTER STRING HANDLING

```
MATCH (svar) TO (svar)
MOVE (svar) TO (svar)
MOVE (svar) TO (nvar)
MOVE (nvar) TO (svar)
APPEND (svar) TO (svar)
CMOVE (sval) TO (svar)
CMATCH (sval) TO (sval)
BUMP (svar)
BUMP (svar) BY (nlit)
RESET (svar) TO (sval)
RESET (svar) TO (nvar)
RESET (svar)
ENDSET (svar)
LENSET (svar)
CLEAR (svar)
```

```
      EXTEND (svar)
      CHAIN (svar)
      LOAD (svar) FROM (nvar) OF (svar list)
      STORE (svar) INTO (nvar) OF (svar list)
      CLOCK TIME TO (svar)
      CLOCK DAY TO (svar)
      CLOCK YEAR TO (svar)
      TYPE (svar)

ARITHMETIC

      ADD (nvar) TO (nvar)
      SUB (nvar) FROM (nvar)
      MULT (nvar) BY (nvar)
      DIV (nvar) INTO (nvar)
      MOVE (nvar) TO (nvar)
      COMPARE (nvar) TO (nvar)
      LOAD (nvar) FROM (nvar) OF (nvar list)
      STORE (nvar) INTO (nvar) OF (nvar list)

INPUT/OUTPUT

      KEYIN (list)
      DISPLAY (list)
      CONSOLE (list)
      BEEP
      PRINT (list)
      RELEASE
      PREPARE n,(svar)
      OPEN n,(svar)
      CLOSE n
      WRITE n,RN;(list)
      WRITE n,RN;(list);
      WRITAB n,RN;(list)
      WRITE n,SEQ;(list)
      WRITE n,SEQ;(list);
      WEOF n,RN
      WEOF n,SEQ
      READ n,RN;(list)
      READ n,RN;(list);
      READ n,SEQ;(list)
      READ n,SEQ;(list);
```

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.
*+	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.
*<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 20 seconds for KEYIN statement.

APPENDIX C

PROGRAM EXAMPLES

Simple ANSWER Program

```
. SIMPLE ANSWER PROGRAM
.
PORTN    FORM "4"
IDCODE   DIM 9
ID       INIT "DATAPOINT"
.
         CLOSE 1
         CLOSE 2
         CLOSE 3
         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
         CLOSE 1
         CLOSE 2
         CLOSE 3
         CONSOLE "MASTER",PORTN
LOOP     KEYIN *N,*EL,"PROGRAM NAME: ",FILNAM
         TRAP NONAME IF CFAIL
         CHAIN FILNAM
NONAME   DISPLAY "*** NO SUCH PROGRAM ***"
         GOTO LOOP
```

```
. DATASHARE ANSWER PROGRAM
.
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
ZERO     FORM     "0"
ONE      FORM     "1"
FOUR     FORM     "4"
EIGHT    FORM     "8"
TEN      FORM     "10"
N28      FORM     "28"
NFEB     FORM     "29"
N30      FORM     "30"
N31      FORM     "31"
SPACE    INIT     " "
CENT     INIT     ", 19"
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
SJAN     INIT     "JANUARY"
SFEB     INIT     "FEBRUARY"
SMAR     INIT     "MARCH"
SAPR     INIT     "APRIL"
SMAY     INIT     "MAY"
SJUN     INIT     "JUNE"
SJUL     INIT     "JULY"
SAUG     INIT     "AUGUST"
SSEP     INIT     "SEPTEMBER"
SOCT     INIT     "OCTOBER"
SNOV     INIT     "NOVEMBER"
SDEC     INIT     "DECEMBER"
LINE     DIM      100
SYSFILE  INIT     "SYSFILE"
DAYFILE  INIT     "DAYFILE"
.
         CLOSE    1
         CLOSE    2
         CLOSE    3
         DISPLAY  *ES,*N,"D A T A S H A R E    PORT ",PORTN;
         OPEN     1,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      FOUR INTO NYEAR1
         MULT     FOUR BY NYEAR1
         COMPARE  NYEAR1 TO NYEAR2
         GOTO     LEAP IF EQUAL
         MOVE     N28 TO NFEB
LEAP     SUB      N31 FROM NDAY1
         GOTO     JAN IF LESS
         GOTO     JAN IF EQUAL
         SUB      NFEB FROM NDAY1
         GOTO     FEB IF LESS
         GOTO     FEB IF EQUAL
         SUB      N31 FROM NDAY1
         GOTO     MAR IF LESS
         GOTO     MAR IF EQUAL
         SUB      N30 FROM NDAY1
         GOTO     APR IF LESS
         GOTO     APR IF EQUAL
         SUB      N31 FROM NDAY1
         GOTO     MAY IF LESS
         GOTO     MAY IF EQUAL
         SUB      N30 FROM NDAY1
         GOTO     JUN IF LESS
         GOTO     JUN IF EQUAL
         SUB      N31 FROM NDAY1
         GOTO     JUL IF LESS
         GOTO     JUL IF EQUAL
         SUB      N31 FROM NDAY1
         GOTO     AUG IF LESS
         GOTO     AUG IF EQUAL
         SUB      N30 FROM NDAY1
         GOTO     SEP IF LESS
         GOTO     SEP IF EQUAL
         SUB      N31 FROM NDAY1
         GOTO     OCT IF LESS
         GOTO     OCT IF EQUAL
         SUB      N30 FROM NDAY1
         GOTO     NOV IF LESS
         GOTO     NOV IF EQUAL
         MOVE     SDEC TO DATE
         GOTO     START1
NOV      ADD      N30 TO NDAY1
         MOVE     SNOV TO DATE
         GOTO     START1
OCT      ADD      N31 TO NDAY1
         MOVE     SOCT TO DATE
         GOTO     START1
SEP      ADD      N30 TO NDAY1
         MOVE     SSEP TO DATE
```

```
        GOTO     START1
AUG     ADD      N31 TO NDAY1
        MOVE     SAUG TO DATE
        GOTO     START1
JUL     ADD      N31 TO NDAY1
        MOVE     SJUL TO DATE
        GOTO     START1
JUN     ADD      N30 TO NDAY1
        MOVE     SJUN TO DATE
        GOTO     START1
MAY     ADD      N31 TO NDAY1
        MOVE     SMAY TO DATE
        GOTO     START1
APR     ADD      N30 TO NDAY1
        MOVE     SAPR TO DATE
        GOTO     START1
MAR     ADD      N31 TO NDAY1
        MOVE     SMAR TO DATE
        GOTO     START1
FEB     ADD      NFEB TO NDAY1
        MOVE     SFEB TO DATE
        GOTO     START1
JAN     ADD      N31 TO NDAY1
        MOVE     SJAN TO DATE
START1  ENDSET   DATE
        MOVE     NDAY1 TO DAY
        COMPARE  TEN TO NDAY1
        GOTO     START2 IF NOT LESS
        BUMP     DAY
START2  APPEND   DAY TO DATE
        APPEND   CENT 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     2,DAYFILE
        MOVE     ZERO TO RN
LOOP0   READ     2,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  LENSET   LINE
        RESET    LINE
        DISPLAY  *+,LINE
```

```
        ADD       ONE 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      1,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       ONE FROM PORTN
        WRITE     1,PORTN;RN,DATE,TIME
        CLOSE     1
        STOP
.
NEXTID  ADD       FOUR TO RN
        GOTO      LOOP2
.
IDFAIL  BEEP
        DISPLAY   "*** INVALID ID ***"
        SUB       ONE FROM IDCTR
        GOTO      LOOP1
KABOOM  CONSOLE   *P60:1,*EL,"ID OVERRUN"
        BEEP
        DISPLAY   "*** INVALID ID ***"
        GOTO      LOOP1
```

```
. DATASHARE MASTER PROGRAM
.
PORTN    FORM     "3"                THE NUMBER OF THIS PORT
SYSFILE  INIT     "SYSFILE"
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"
.
         CLOSE    1
         CLOSE    2
         CLOSE    3
         RELEASE
         CONSOLE  "MASTER",PORTN," "
         DISPLAY  *ES
         OPEN     1,SYSFILE
         SUB      ONE FROM PORTN
         READ     1,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    1
         CHAIN    CMDLIN
CFAIL    OPEN     1,SYSFILE
         KEYIN    *N,"WHAT?",*N,CMDLIN
         GOTO     TRYAGN
.
GETNAM   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
.
HELPI    KEYIN    *ES,*N:
                  "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
.
HELLOI   CALL     GETNAM
         MOVE     CMDLIN TO LINITM
         MOVE     EIGHT TO RNX
HELLO2   READ     1,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     1,PORTN;RN,LINE
         WRITE    1,PORTN;RNX,LINE
         MOVE     RNX TO RN
```

```
        GOTO    CMDREQ
.
NEXTID  ADD     FOUR 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    1,RNX;LINE
        RESET   LINE TO 11
        MOVE    NINE TO COUNT
        GOTO    CATR1
CATR    READ    1,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    1,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     1,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
```