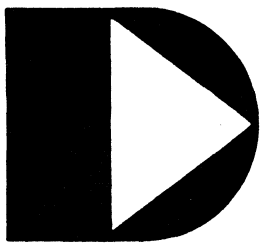
DOS DATABUS COMPILER DBCMP User's Guide

Version 2

February, 1977

Model Code No. 50182

DATAPOINT CORPORATION



The leader in dispersed data processing ™

COPYRIGHT® 1976 BY DATAPOINT CORPORATION. PRINTED IN U.S.A.

DOS. DATABUS COMPILER
DBCMP

User's Guide

Version 2

February, 1977

Model Code No. 50182

DATAPOINT CORPORATION

April 1, 1977

Addendum to: DBCMP User's Guide
Version 2
February, 1977
50182

Reference: Chapter 8, after the second paragraph.

Add:...
"When the result causes the OVER flag to be set the LESS,ZERO flags are indeterminate."

Reference: Chapter 8, section 8.1, at the bottom of page 8-2.

Delete:...
", ZERO"

Add:
"The LESS,ZERO flags are indeterminate."

9725 DATAPOINT DRIVE SAN ANTONIO, TEXAS 78284 (512) 699-7000

Reference: Chapter 8, section 8.2, in the middle of page 8-4.

Change:
"The following flag will be set: OVER, LESS"

To:
"The following flags will be set: OVER"

Add:
"The LESS,ZERO flags are indeterminate."

Reference: Chapter 8, section 8.3, at the bottom of page 8-5.

Delete:...
", LESS"

Add:...
"The LESS,ZERO flags are indeterminate."

Reference: Chapter 8, section 8.4, in the middle of page 8-8.

Delete:...
", ZERO"

Add:...
"The LESS,ZERO flags are indeterminate."

Delete:...
", ZERO"

Add:...
"The LESS,ZERO flags are indeterminate."

Reference: Chapter 8, section 8.5, in the middle of page 8-9.

Add:...
"The LESS,ZERO flags are indeterminate."

Reference: Chapter 8, section 8.6, on page 8-10.

Change:...
"The LESS and ZERO...."

To:...
"The LESS, OVER and ZERO..."

Delete:...
"-- Since the result is not moved to the destination variable, the format of the result is not taken inot consideration when setting the condition flags. This means that the OVER condition flag can never be set by the COMPARE instruction."

Reference: Chapter 15, section 15.7, at the bottom of page 15-11.

Replace:
"-- c) Since UPDATE modifies logical records instead of physical records, it is possible to tab across pyhsical record boundaries."

With:
"-- c) It is an illegal operation to follow an UPDATE with a DELETE. This operation can destroy your file."

Reference: Chapter 15, section 15.9, at the top of page 15-14.

Add:...
"-- It is an illegal operation to follow an UPDATE with a DELETE. This operation can destroy your file."

CHAPTER 8. ARITHMETIC INSTRUCTIONS

The arithmetic instructions are used to perform the various arithmetic operations upon DATABUS operands. Generally all arithmetic instructions have the following form:

<label> <oper> <soper><prep><doper>

where: <label> is an execution label.
<oper> is the DATABUS arithmetic operation.
<soper> is the source operand.
<prep> is a valid preposition.
<doper> is the destination operand.

The DATABUS operation is performed using the source and destination operands. The result of the operation is generally transferred to the destination operand. The content of the source operand is never modified. When the result causes the OVER flag to be set the LESS,ZERO flags are indeterminate.

8.1 ADD

The ADD instruction causes the content of source operand to be added to the content of destination operand. The result (sum) is placed in the destination operand. This instruction has the following formats:

1) <label> ADD <snvar><prep><dnvar>
2) <label> ADD <nlit><prep><dnvar>

Where: <label> is an execution label (see section 2.).
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The source numeric operand is never modified.

-- <dnvar> contains the result (sum) of the ADD.

-- The flags OVER, LESS, ZERO (or EQUAL), are set appropriately.

-- The rounding and truncation rules are applicable (see section 2.7).

Examples:

```
        X       FORM    "123.45"
        Y       FORM    "267.22"

                ADD     X TO Y

                Y will contain 390.67
                The following flag(s) will be set: None
```

Example:

```
        CAT     FORM    "100.50"

                ADD     ".005" TO CAT

                CAT will contain 100.51
                The following flag(s) will be set: None
```

Example:

```
        NUM     FORM    "-245.0000"
        NUM2    FORM    "800.0"

                ADD     NUM TO NUM2

                NUM2 will contain 555.0
                The following flag(s) will be set: None
```

Example:

```
        N       FORM    "00.0"

                ADD     "100.00" TO N

                N will contain 00.0
                The following flag(s) will be set: OVER
                The LESS,ZERO flags are indeterminate.
```

```
C1      FORM    "5.60"
C2      FORM    "1.665"

        SUB     C2 FROM C1

        C2 will contain 3.94
        The following flags will be set: None
```

Example:

```
NUMBR   FORM    "-345"

        SUB     "700.5" FROM NUMBR

        NUMBR will contain 1045
        The following flags will be set: OVER
        The LESS,ZERO flags are indeterminate.
```

Example:

```
Y1      FORM    " 10.00"
Y2      FORM    " 20.005"

        SUB     Y2 FROM Y1

        Y2 will contain -10.01
        The following flags will be set: LESS
```

8.3 MULTIPLY (MULT)

The MULT instruction (the compiler will also accept a mnemonic of MULTIPLY) causes the content of the source numeric (multiplicand) to be multiplied by the contents of the destination numeric operand (multiplier). The result (product) is placed in the destination numeric operand. The instruction has the following formats:

```
1)   <label>  MULT      <snvar><prep><dnvar>
2)   <label>  MULTIPLY  <snvar><prep><dnvar>
3)   <label>  MULT      <nlit><prep><dnvar>
4)   <label>  MULTIPLY  <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- The execution label <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The flags OVER, LESS, ZERO (or EQUAL) are applicable.

-- The source numeric operand is not modified.

-- The destination numeric operand contains the result (product).

-- The sum of the number of characters in the source operand and the destination operand must not exceed 31. (The compiler does not check this limit. If it is exceeded the interpreter will produce erroneous results.)

-- The truncation and rounding rules are applicable.

Example:

```
        M1      FORM    "010"
        M2      FORM    "012"

                MULT    M1 BY M2

                M2 will contain 120
                The following flag(s) will be set: None
```

Example:

```
        X123    FORM    "12000.00"

                MULT    "1.1" BY X123

                X123 will contain 13200.00
                The following flag(s) will be set: None
```

Example:

```
        NEG     FORM    "-10.5"

                MULT    "10" BY NEG

                NEG will contain 105.0

                The following flag(s) will be set: OVER
                The LESS,ZERO flags are indeterminate.
```

Example:

```
        ZERO    FORM    "000"
        N       FORM    "155.00"

                DIV     ZERO INTO N

                N will contain 999.99
                The following flag(s) will be set: OVER
```

Example:

```
        ZERO    FORM    "00.00"
        N       FORM    "155.00"

                DIV     ZERO INTO N

                N will contain __.00
                The following flag(s) will be set: OVER
                The LESS,ZERO flags are indeterminate.
```

Example:

```
        N1      FORM    "100"

                DIV     "0.1" INTO N1

                N1 will contain __0
                The following flag(s) will be set: OVER
                The LESS,ZERO flags are indeterminate.
```

8.5 MOVE

The MOVE instruction causes the content of the source numeric operand to replace the content of the destination numeric operand. The instruction has the following formats:

```
1)   <label>  MOVE      <snvar><prep><dnvar>
2)   <label>  MOVE      <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The contents of the source numeric operand is never modified.

-- The destination numeric variable contains the result of the MOVE operation.

-- The OVER, LESS, ZERO (or EQUAL) flags are applicable.

-- The truncation and rounding rules are applicable.

Example:

```
        SOURCE  FORM    "12345"
        DESTIN  FORM    6.2

                MOVE    SOURCE TO DESTIN

                DESTIN will contain _12345.00
                The following flag(s) will be set: None
```

Example:

```
        D1      FORM    4.2
                MOVE "12345" TO D1

                D1 will contain 2345.00
                The following flag(s) will be set: OVER
                The LESS,ZERO flags are indeterminate.
```

Example:

```
        S       FORM    "12345.51"
        D       FORM    "99999"

                MOVE    S TO D

                D will contain 12346
                The following flag(s) will be set: None
```

Example:

```
        N       FORM    "999.99"

                MOVE    "0.0" TO N
```

N will contain ___.00
The following flag(s) will be set: ZERO

8.6 COMPARE

The COMPARE instruction is used to compare two numeric quantities. The instruction has the following formats:

1) <label> COMPARE <snvar><prep><dnvar>
2) <label> COMPARE <nlit><prep><dnvar>

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> is a valid numeric literal.

-- The contents of the source numeric operand are never modified.

-- The contents of the destination numeric variable are never modified.

-- The LESS, OVER and ZERO (or EQUAL) condition flags are set exactly as if a SUBTRACT instruction had been executed instead of a COMPARE.

-- Rounding takes place when the COMPARE instruction is executed.

Example:

```
ONEH    FORM    "100.00"

        COMPARE "100" TO ONEH

        The following flag(s) will be set: ZERO (EQUAL)
```

15.7 UPDATE

The UPDATE instruction allows tabbing while modifying an indexed record. UPDATE allows characters to be written into any character position of an indexed record without disturbing the rest of the record. This instruction may have one of the following general formats:

```
1)   <label>  UPDATE    <ifile>;<list>
2)   <label>  UPDATE    <rifile>;<list>
```

where: <label> is an execution label (see section 2.).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- UPDATE is used to modify the last indexed record accessed by any indexed record instruction (typically a READ or READKS).

-- With the following exceptions UPDATE functions the same as WRITAB.

   a) All tab positions are calculated relative to the beginning of the logical record, rather than relative to the beginning of the physical record.

   b) The initial position within the data file is determined as described above, rather than being furnished by a variable.

-- It is an illegal operation to follow an UPDATE with a DELETE. This operation can destroy your file.

-- Attempting an UPDATE when no other index operation has been performed prior to the execution of the UPDATE, will cause and I/O error.

-- It is possible to overstore the 015 (logical end of record) and the 003 (physical end of record) characters when using UPDATE. If extreme care is not exercised, this can result in more than one record being turned into a single very large

-- It is illegal operation to follow an UPDATE with a DELETE. This operation can destroy your file.

-- <label> is optional.

-- The logical string of <svar> specifies the key to be deleted.

-- One DELETE must be executed for each index file which will need the key deleted.

-- If the key is null, an I/O error will result.

-- If the key cannot be found in the index, the OVER condition flag is set.

-- The indexed record is deleted by overstoring every character in the record with an 032 (octal). This includes the logical end of record character (015).

-- Both the DOS REFORMAT utility and the Databus interpreters ignore all 032 characters while reading. Therefore, while reading these characters do not appear to exist.

-- The DOS REFORMAT utility may be used to eliminate the 032 control characters from the data file.

-- If the indexed record to be deleted has already been deleted, then the only action taken is to delete the key from the index file.

PREFACE

The DOS DATABUS Compiler (DBCMP) is the compiler to be used to compile DATABUS programs for the DATASHARE 3, DATASHARE 4, and DATABUS 11 interpreters. This compiler is compatible with all current Datapoint DOS releases. This manual provides the reference material required by users of the DATABUS language. It is designed to be used as reference only, and not as a tutorial.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

The DATABUS language is an interpretive, high level language designed for business applications. It has been designed to run under the Datapoint Disk Operating System and takes advantage of all of its file handling capabilities (dynamic file allocation, random or sequential files, and the powerful Indexed Sequential Access Method).

Verbs are provided to permit simple yet flexible operator interaction with the program, thus enabling levels of data entry and checking ranging from simple keypunch to extremely sophisticated intelligent data entry. A complete set of string manipulation verbs are available, along with a flexible arithmetic package. An extensive set of file manipulation verbs complete a powerful business-oriented language.

The complete DATABUS language may not be compatible with all DATASHARE 3, DATASHARE 4, and DATABUS 11 Interpreters. The following is a brief description of the current DATASHARE 3, DATASHARE 4, and DATABUS 11 interpreters. Refer to the appropriate user's guide for more detailed information about the interpreters.

DS3A3360	DATASHARE 3 Interpreter supporting up to eight 3360 terminals on a 2200 DOS.A system.
DS3A3600	DATASHARE 3 Interpreter supporting up to eight 3600 terminals on a 2200 DOS.A system.
DS3B3360	DATASHARE 3 Interpreter supporting up to eight 3360 terminals on a 2200 DOS.B system or a 2200 DOS.A system with a 4K disk controller.
DS3B3600	DATASHARE 3 Interpreter supporting up to eight 3600 terminals on a 2200 DOS.B system or a 2200 DOS.A system with a 4K disk controller.
PSDS3	DATASHARE 3 Interpreter supporting up to sixteen 3360 or 3600 terminals on a 5500 DOS.D or DOS.E system. (This interpreter will execute in either partition of the Partition Supervisor.)

DS42200	DATASHARE 4 Interpreter supporting up to four 3360 terminals on a 2200 DOS.A or DOS.B system.
DS42200X	DATASHARE 4 Interpreter supporting up to four 3600 terminals on a 2200 DOS.A or DOS.B system.
DS45500	DATASHARE 4 Interpreter supporting up to 16 3360 or 3600 terminals on a 5500 DOS.D or DOS.E system. In addition this interpreter can be configured to support remote diskette stations via dial-up telephone lines in a networking configuration (replacing DS3NET). It may also be configured for external communications through a MULTILINK configuration.
DB11	DATABUS 11 Interpreter executing DATABUS code programs from the processor console on a 2200, Diskette 1100, or 5500, DOS.A, DOS.B, DOS.C, DOS.D, or DOS.E systems.
DBML11	DATABUS MULTILINK 11 interpreter executing two DATABUS code programs. The primary program was the processor console and the secondary (or utility) program may be used for utility functions. Internal (between primary and secondary program) and external (with a remote or host processor) communications are support. The interpreter executes on a Datapoint 1150 DOS.C system.

CHAPTER 2. STATEMENT STRUCTURES

There are four basic types of statements in the DATABUS language: comment, compiler directive, data area definition and program execution. All of the statements (except comments) use the following basic format:

<label> <operation> <operands> <comment>

where: each of the fields above is separated from the others by at least one space,
<label> is a letter, followed by any combination of up to seven letters and digits, (this does not include special characters),
<operation> denotes the operation to be performed on the following operands,
<operands> are any operands required by the <operation>, and
<comment> is any comment the user wants to make about the instruction or about program execution.

The label field is considered empty if a space appears in the first column of the line. The following are examples of valid labels:

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

The following are examples of invalid labels:

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

The compiler keeps track of two distinct sets of labels; data labels and execution labels. Data labels are those present on data area definition statements. Execution labels are those labels used by the program control instructions (see chapter 6.) to alter the normal flow of program execution.

Data labels must be unique among themselves; that is, no data label can be the same as any other data label. Execution labels

must also be unique among themselves. However, a label may be used both as a data label and also as an execution label.

Although there are exceptions (for more details see the sections that describe the instructions individually), the operand field for most of the instructions has the following general format:

<source operand><separator><destination operand>

where: <source operand> is the first operand required by the operation,
<destination operand> is the second operand required by the operation, and
<separator> must be a comma or a valid preposition.

If a comma is used as the separator it cannot be preceded by any spaces, but may be followed by any number of spaces (including none). The prepositions that may be used as separators are BY, TO, OF, FROM, USING, WITH, or INTO. If one of these prepositions is used as the separator, it must be preceded and followed by at least one blank. Note that any of these preposition may be used even if it does not make sense in English.

The following are all examples of valid statements:

```
LABEL1   ADD      PCS TO TOTAL
LABEL2   ADD      PCS OF TOTAL          THIS IS A COMMENT
LABEL3   ADD      PCS, TOTAL
LABEL4   ADD      PCS,TOTAL
LABEL5   ADD      PCS     TO     TOTAL
```

The following are examples of invalid statements:

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

Some of the operations require a list of items in the operand field. Such a list is typically made up of variable names, literals, and list controls separated by commas. 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. Comments may be included after the colon used for continuation.

For example, the two statements:

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

will perform the same function.

2.1 Comments

Comment lines have a period, asterisk, or plus sign in the first column, and may appear anywhere in the program. Comments are useful in making it easier for someone reading through the program to understand program logic, subroutine function, subroutine parameterization, etc.

Comments that begin with a period are simply copied from the source program to any listing requested by the user.

Comments that begin with an asterisk are treated like comments that begin with a period, unless there are fewer than 12 lines at the bottom of the current page. If there are fewer than 12 lines, comments that begin with an asterisk will be printed at the top of the next page. This allows comments to appear on the same page as the program instructions that are being described by the comments. Use of the asterisk at the beginning of each section or subroutine description is encouraged since this greatly enhances program readability.

Comments that begin with a plus sign will always be printed at the top of the next page. This allows major sections of the program to be started at the top of a page. The plus sign should be used cautiously, since it can easily waste great quantities of paper.

2.2 Compiler Directives

Compiler directives are provided to make the compilation process easier and more flexible.

There is a compilation directive which allows a programmer to include other files in the current compilation. This directive allows large programs to be broken into several smaller, easier-to-edit files. It also allows a single file to be used for a set of subroutines or data definition blocks which are common to more than one program.

There is also a compilation directive which allows the absolute value of a symbolic name to be defined. A name defined in this manner may then be used for tab positioning in disk I/O statements or cursor positioning in KEYIN, DISPLAY and CONSOLE statements.

2.3 Data Area Definition

The user's data area must be defined by using file declaration or data definition statements. File declaration statements are used to reserve space for the system information needed for all disk accessing, while data definition statements are used to describe the format of any variables used in a program. For information about the size of the user's data area, see the user's guide of the appropriate interpreter. All of these statements must have labels which are used to reference the variable or logical file defined. All labels used with data definition and file declaration statements are data labels (see section 2.).

2.4 Program Execution

The program execution statements are those that actually do the data manipulation and must conform to the following rules:

-- They must appear after any data area definition statements.

-- They may or may not have labels.

-- Any label used on one of these statements is an execution label (see section 2.).

-- Program execution always begins with the first executable statement.

2.5 Literals

Literals are useful when a constant value is needed as one of the operands of an instruction. Using literals will save user's data area.

A literal has one of the following formats:

```
"<string>"
<dnum>
"<char>"
<occ>
```

where: <string> is any sequence of characters with the exceptions described below in the section on the forcing character (#). This string may be either a numeric string (see section 4.1) or a character string (see section 4.2).
<dnum> is a decimal number between -128 and 127.
<char> is any single character. (The forcing character rules do not apply.)
<occ> is an octal control character.

See the sections describing the individual instructions for the format that may be used with those instructions allowing literals.

The following criteria apply to literals with the "<string>" format:

-- The string may be from 1 through 40 characters in length (excluding the quotes).

-- The string must be enclosed in double quotes.

-- When the literal is used as a character string the formpointer is always equal to 1.

-- When the literal is used as a character string the logical length pointer always points to the last character of the literal.

-- Most instructions that make use of these literals require that the literal be the first operand of the instruction (for more details see the sections that describe the instructions individually).

Some examples of instructions that may use literals of the "<string>" format follow:

```
STORE     "APPLES" INTO X OF S1,S2,S3
ROLLOUT   "CHAIN FIX22"
CHAIN     "NEXTPROG"
OPEN      FILE1,"DATAFILE"
```

```
PREPARE   FILE1,"USERDATA"
MOVE      "MESSAGE" TO M3442
MOVE      "100.55" TO VALUE
APPEND    "." TO STR1
MATCH     "YES" TO ANSWER
ADD       "23.46" TO TOTAL
SUBTRACT  "1" FROM COUNT
MULTIPLY  ".1" BY TAX
DIVIDE    "33.3333" INTO FACTOR
COMPARE   "10" TO LINENUMB
```

The following criteria apply to octal control characters:

-- The octal control character must be between 000 and 037, inclusive.

-- The first character of an octal control character must be a zero.

-- Note that some of these octal control characters are used for control purposes in disk files (000, 003, 011, 015) and others are used as control characters in DISPLAY, KEYIN and CONSOLE statements. Improper use of these control characters can result in invalid program execution.

2.6 The Forcing Character

Since the second double quote is used to indicate the end of the string, any literal of the form "<string>" needs a special technique to include a double quote as a character within the <string>. The technique used by the Databus language is to define the pound sign (#) to be a forcing character.

Putting the pound sign within a string tells the compiler that the next character in the string should be included within the string. The character following the pound sign is not checked for any special significance, it is simply picked up and put into the string. The pound sign used as a forcing character is not put into the string. This means that to put the pound sign itself into a string you must do so by using a previous pound sign as a forcing character.

For example,

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

would display exactly:

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

on the screen.

Note that the forcing character convention does not apply to literals of the "<char>" format. <char> may be any character, including the double quote character and the pound sign character. For example,

```
CMOVE     """ TO STRING
```

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

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

would be used to move a double quote sign into the variable STRING.

2.7 Numeric Definitions

The following definitions will be established so that the ensuing discussion in subsequent chapters will be more meaningful.

2.7.1 Integer/Fraction

Numeric String Variables (or literals) are composed of two parts.

a) Integer - The integer portion of a numeric variable is the portion of the numeric string that exists to the left of the decimal point. If the decimal point does not exist explicitly, the decimal point is implied to be to the right of the rightmost digit of the numeric string.

b) Fractional - The fractional portion of a numeric variable

is the portion of the numeric string that exists to the right of the decimal point.

For example consider the following:

```
A       FORM    "123.45"
B       FORM    "678."
C       FORM    "90"
```

A has a value of 123 for the integer portion and 45 for the fractional portion. B has a value of 678 for the integer portion. C has a value of 90 for the integer portion (the decimal point is implied to the right of the zero).

2.7.2 Rounding/Trucation

When the result of an arithmetic operation consists of more characters than can be contained in the destination variable, the result is truncated, rounded or both truncated and rounded so that it will "fit" in the destination variable.

Truncation is the process of eliminating those characters that will not fit in the destination variable. Truncation may occur either on the right or on the left. Right truncation means some of the least significant digits of the result are lost, while left truncation means that some of the most significant characters are lost. Usually, the arithmetic instruction that causes left truncation of the result will set the OVER condition flag to indicate arithmetic overflow.

Rounding is a modified form of right truncation. For details on rounding, see section 2.7.3. Unless specificly mentioned otherwise, rounding will be used instead of right truncation.

The following rules are used to determine which characters will be lost if truncation or rounding is necessary:

a) If the destination variable is defined to contain a decimal point, the result (of the arithmetic operation) is aligned so that its decimal point will overstore the destination variable's decimal point. Any characters that will not fit after this alignment are lost.

b) If the destination variable is defined without a decimal point, alignment occurs as if there were a decimal point just after the least significant digit of the destination variable.

2.7.3 Rounding Rules

To determine when rounding is necessary, see section 2.7.2. The following rules should be used to distinguish between right truncation and rounding. To understand the following rules the distinction between the rounding digit and the rounded digit must be clear. The rounding digit is the most significant of the digits lost when rounding a number, while the rounded digit is the least significant of the digits that are not lost.

a) If the rounding digit is a digit from 0 to 4, then the rounded digit remains unchanged.

b) If the rounding digit is the digit 5:

    1) If the rest of the digits that will be lost are zero (0):

        a. If the result (of the arithmetic operation) is a negative number, then the rounded digit remains unchanged.

        b. If the result (of the arithmetic operation) is a positive number, then the rounded digit is incremented by one (1).

    2) If any of the rest of the digits that will be lost are non-zero, then the rounded digit is incremented by one (1).

c. If the rounding digit is a digit from 6 to 9, then the rounded digit is incremented by one (1).

2.8 Character String Definitions

The following terms will be used in the description of character string variables.

character string variable -- made up of four parts; the logical length pointer, the formpointer, the physical string and the ETX.

| llp | fp | physical string | ETX |

physical string -- made up of three parts; the prefix, the (logical) string and the suffix.

```
¦ prefix ¦ (logical) string ¦ suffix ¦
```

logical string -- the string usually modified by the instructions. It is defined by the formpointer and the logical length pointer. The first character in the logical string is the head (the character pointed to by the formpointer). The last character in the logical string is the tail (the character pointed to by the logical length pointer).

```
¦ head ¦          ¦ tail ¦
```

null string -- a string with the formpointer and the logical length pointer both set to zero.

2.9 A Sample Program

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

CHAPTER 3. COMPILER DIRECTIVES

Two directives are available to give the user more control over the compilation process. One is the EQU statement and the other is the INCLUDE statement.

3.1 EQUATE (EQU)

The EQU statement allows a label to be assigned a decimal numeric value from 1 through 249.

This is particularly useful when one defines the format of disk records to be used in a data base. If all item positions within the record are defined using the EQU directive, then changes in item positions can be achieved by simply changing the one directive value. If the EQU were not used, changing the record format would mean changing all disk I/O statements that depend on this format. The user would have to hunt through all programs using this format to change all disk I/O statements to conform to the new record format.

The general format of the EQU statement is as follows:

```
<label>  EQU      <dnum>
```

where: <label> is a data label (see section 2.)
<dnum> is the decimal number to be substituted for any occurance of the label within the program being compiled.

For example:

```
LM       EQU      5
```

A label which is defined in this manner may be used as tab values in disk I/O statements and as cursor positions in KEYIN, DISPLAY, and CONSOLE statements.

3.2 INCLUDE (INC)

This statement allows another text file to be included, at the point where the INCLUDE statement appears, as if the lines actually existed in the main file being compiled. Note that the INCLUDE directive can be used to include a file containing any EQU directives and data variable definitions which are needed to define the record format of a data base. This allows the programmer to enter the information about the data base into only one file instead of entering it into every program that needs to know about the data base. Modification of the format also becomes easier, since the programmer need modify only one file before compiling all of the programs again.

The INCLUDE statement can have one of the following formats:

```
<label>  INCLUDE   <DOS file specification>
<label>  INC       <DOS file specification>
```

where: <label> is a data label (see section 2.).
<DOS file specification> is a DOS compatible specification of the file to be included in the program.

Programming Considerations:

-- Including a file will cause all of the lines in that file to be scanned as if they existed in place of the INCLUDE line.

-- The assumed extension on included files is TXT but may be specified to be any extension.

-- If no drive is specified, all drives starting with drive zero will be scanned for the file.

-- Inclusions may be nested up to four deep, with a maximum of 16 included files.

For example:

```
INC       RECDEFS
```

would cause all of the lines from file RECDEFS/TXT to be scanned as if they existed instead of the INC statement.

CHAPTER 4. DATA DEFINITION

There are two types of data used within the DATABUS 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.

Whenever a data variable is to be used in a program, it must be defined at the beginning by using one of the data definition statements. The data definition statements reserve space in the user's data area for the data variable whose name is given in the label field. (This space is always reserved using one of the formats described below.) Note that all variables must be defined before the first executable statement in the program and that once an executable statement is given, no more variables may be defined.

4.1 Numeric String Variables

Numeric strings have the following memory format:

```
octal ascii ascii ascii ascii octal
 0200   1     2     .     3    0203
```

The leading character (0200) is used as an indicator that the string is numeric. The trailing character (0203) is used to indicate the location of the end of the string (ETX).

Programming Considerations:

-- The format of a numeric string is set at definition time and does not change throughout the execution of the program.

-- Negative numbers are represented by using one of the characters before the decimal point for a minus sign.

-- The physical length of a numeric string is limited to 21 characters (including the decimal point and minus sign, but excluding the 0200 and 0203 characters).

-- Numeric items always keep their proper format internally.

-- To be a valid numeric string, the following must be true.

   a. Spaces are acceptable only when they are leading spaces.

   b. Only one minus sign is allowed.

   c. The minus sign must be next to the most significant character.

   d. Only one decimal point is allowed.

   e. Except for the cases mentioned above, only digits are allowed.

   f. A string made up of any combination of spaces, decimal points and minus signs without at least one digit is not allowed.

   -- Whenever a new value is assigned to a numeric variable, it is reformatted to have the format of that variable.

4.2 Character String Variables

Character strings have the following memory format:

```
oct oct asc asc asc asc asc asc asc asc asc asc asc asc oct
011 005  T   H   E       B   R   O   W   N       F   O   X  0203
```

The first byte is called the logical length pointer and points to the last character currently being used in the string (N in the above example). The second byte is called the formpointer and points to the first character currently being used in the string (B in the above example). The use of the logical length pointer and the 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 through which the programmer deals with individual characters within the string.

Programming Considerations:

-- The term physical length will be used to mean the number of possible data characters in a string (13 in the above example).

-- The physical length of string variables is limited to 127.

-- The logical length pointer will never be greater than the physical length of the string.

-- The formpointer will always be between zero and the logical length pointer.

-- A zero formpointer indicates a null string.

-- In the case of character string variables, the actual amount of user's data area reserved is three bytes greater than the physical length of the variable.

4.3 Common Data Areas

Since the interpreter 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, (preferably at the beginning of each program). The point in this is to cause each common variable to occupy the same locations in each program. Strange results in program execution usually occur if a common variable is misaligned with respect to the variable in the previous program.

b. For the first program to use the variables, define them in the normal way. Then, for each succeeding program, 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 WON'T BE LOADED"
```

File declarations may not be made common between programs. Mis-alignment in file declarations could easily cause catastrophic destruction of the file structure under DOS. Therefore, whenever a program is loaded, all logical files are initialized to being closed and must be opened before any file I/O can occur. When chaining between programs, one should always close all files in

which new space could have been allocated and then re-open the files in the next program.

4.4 FORM

This instruction is used to define numeric string variables. They may be defined using one of the formats shown below:

```
1)   <label>  FORM     <dnum1>.<dnum2>
2)   <label>  FORM     <dnum1>.
3)   <label>  FORM     .<dnum2>
4)   <label>  FORM     <dnum1>
5)   <label>  FORM     <nlit>
```

where: <label> is a data label.
<dnum1> is a decimal number indicating the number of digits that should precede the decimal point.
<dnum2> is a decimal number indicating the number of digits that should follow the decimal point.
<nlit> is a literal of the form "<string>" (see section 2.5).

Programming Considerations:

-- <nlit> must be a valid numeric string (see section 4.1).

-- The initial value of variables defined using formats (1), (2), (3) and (4) above will be zero.

-- A decimal point will be included as part of any value assigned to variables defined using formats (1), (2) and (3) above.

-- The initial value of a variable defined using format (4) above will be the value of the numeric string between the quotes. (A decimal point found between the quotes will be included as part of the initial value.)

-- The number of digits preceding the decimal point of a variable defined using format (5) above, will be the same as the number of digits preceding the decimal point in <nlit>.

== The number of digits following the decimal point of a variable defined using format (5) above, will be the same as the number of digits following the decimal point in <nlit>.

Examples:

```
FRACPART FORM      0.1
RATE     FORM      4.3
AMOUNT   FORM      " 382.40 "
```

In these examples, the FORM instruction used to define RATE will reserve space for four places before the decimal point, the decimal point itself, and three places after the decimal point. RATE can have as its value a numeric string which can cover the range from 9999.999 to -999.999. The value of RATE will be initialized to zero.

The FORM instruction used to define AMOUNT will reserve space for four places before the decimal point, the decimal point itself, and three places after the decimal point. AMOUNT can have as its value a numeric string which can cover the range from 9999.999 to -999.999. The value of AMOUNT will be initialized to 382.400.

4.5 DIM

This instruction is used to define character string variables. They may be defined using the format shown below:

```
<label>  DIM      <dnum>
```

where: <label> is a data label (see section 2.).
<dnum> is a decimal number indicating the number of characters to be reserved for the variable.

Programming Considerations:

-- All of the characters of a variable defined with a DIM statement will be initialized to spaces (octal 040).

-- The formpointer and logical length pointer will be initialized to zero to indicate a null string.

Example:

```
STRING   DIM      25
```

STRING will be defined to have a physical length of 25 and will consume 28 bytes of the user's data area.

4.6 INIT

This instruction is used to define character string variables with an initial value. They may be defined using one of the formats shown below:

1) <label> INIT <slit>
2) <label> INIT <list>

where: <label> is a data label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<list> is any combination of <slit> and <occ> (see section 2.5) elements separated by commas.

Programming Considerations:

-- <slit> must be a valid character string (see section 4.2).

-- The characters in the variable will be initialized to the string appearing between the quotes.

-- The formpointer will point to the first character of the string.

-- The logical length pointer will point to the last character of the string.

-- Use of a colon for continuation of the statement on the next line is not allowed.

Examples:

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

TITLE will be defined to have a physical length of 15 bytes and will consume 18 bytes of user's data area. The formpointer will be set to 1 (pointing to the P) and the logical length pointer will be set to 15 (pointing to the M).

```
TITLE    INIT      "PAYROLL PROGRAM",015,"A,B,C"
```

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 characters A, comma, B, comma, C.

The octal control character 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. More importantly, these characters are used as control characters in DISPLAY, KEYIN, and CONSOLE statements; and improper use of these characters in such statements can result in invalid program execution.

4.7 COMLST

This instruction is used to reserve space in the user's data area to contain information for a RECV or SEND DATABUS instruction. The general format of the statement is:

```
<label>  COMLST    <dnum>
```

where: <label> is a data label.
<dnum> is a decimal number between 1 and 64. This number specifies the maximum number of variables that may appear in a SEND or RECV instruction referencing this COMLST variable.

Programming Considerations:

-- <dnum> must be a decimal number between 1 and 64 inclusive. A <dnum> of 5 specifies that space is reserved in the user data area variable to contain information for 5 variables.

-- The space allocated is 8+2*(dnum) bytes. The eight bytes are used to contain status and control information and the 2*(dnum) bytes are used to contain the addresses of the variables (2 bytes each) that may appear in SEND or RECV statement referencing this COMLST.

Example:

```
A        COMLST    5    (reserves 8+2*5= bytes of user data area.)
```

CHAPTER 5. FILE DECLARATION

A file declaration statement defines a logical file by reserving space in the user's data area for the DOS system information about the disk file being used. Note that since logical file information is stored in the user's data area, the user may have any number of logical files active at any one time providing his data area will contain all of the necessary information.

5.1 FILE

This is the instruction which is used to reserve space in the user's data area for files that will be used for physically or randomly sequential accessing. The general format of the statement is as follows:

```
<label>  FILE
```

where: <label> is a data label (see section 2.).

Programming Considerations:

-- The <label> must be used in all disk I/O statements that will reference this particular logical file.

-- Each use of this statement causes 17 bytes of data area to be consumed. This area is used to store:

   a) the 15 bytes used in the DOS logical file table,

   b) a space compression counter, and

   c) a flag indicating that these are physically-random or sequential-access-only files.

Example:

```
INFILE    FILE
```

The label INFILE will be used in all disk I/O statements that are to use this particular logical file.

5.2 IFILE

This is the instruction which is used to reserve space in the user's data area for files that will be used for indexed-sequential file accessing. The general format of the statement is as follows:

```
<label>  IFILE
```

where: <label> is a data label (see section 2.).

Programming Considerations:

-- The <label> must be used in all disk I/O statements that will reference this particular logical file.

-- Each use of this statement causes 26 bytes of data area to be consumed. This area is used to store:

   a) the information that the FILE declaration stores,

   b) three 3-byte pointers for use by the indexed-sequential access method. These pointers point to:

      1. the beginning of the last record accessed (for updating operations),

      2. the next sequential key (for sequential by key accessing), and

      3. information in the DOS R.I.B. of the index file (used in all accessing operations).

Example:

```
ISAMFILE  IFILE
```

The label ISAMFILE will be used in all disk I/O statements which are to use this particular logical file.

5.3 RFILE

This instruction is identical to the FILE declaration except that the RFILE instruction defines a logical file that will reference a file at a remote station instead of at the central station.

5.4 RIFILE

This instruction is identical to the IFILE declaration except that the RIFILE instruction defines a logical file that will reference a disk file at a remote station instead of at the central station.

CHAPTER 6. PROGRAM CONTROL INSTRUCTIONS

The interpreter normally executes statements starting with the first executable statement and sequentially from there. The program control instructions allow this flow of control to be altered. Some of these instructions may be executed conditionally depending on whether a condition flag is set to true or false (see section 6.1).

6.1 Condition Flags

There are four condition flags set by the interpreter: OVER, LESS, ZERO (the mnemonic EQUAL is also accepted), and EOS. These flags are set to true or false, depending on the results of some of the instructions. For more details on which flags are set and when they are set, see the sections that describe the instructions individually.

6.2 GOTO

The GOTO statement causes the flow of program control to jump to the place in the program indicated in the GOTO statement. The format of the statement may be one of the following:

```
1)   <label1> GOTO      <label2>
2)   <label1> GOTO      <label2> IF <flag>
3)   <label1> GOTO      <label2> IF NOT <flag>
```

where: <label1> is an execution label (see section 2.).
<label2> is an execution label.
<flag> is one of the condition flags (see section 6.1).

Programming Considerations:

-- <label1> is optional.

-- <label2> must be a label on the executable statement where program control is to be transfered.

-- The condition flags are unchanged by the execution of this statement.

-- A GOTO statement with format (2) will transfer control (to the

statement with <label2>) only if the specified condition flag is set to true; otherwise, program control continues in a sequential fashion.

-- A GOTO statement with format (3) will transfer control only if the specified condition flag is set to false.

Example:

```
GOTO      CALC
```

causes control to be transferred to the instruction labeled CALC.

Example:

```
GOTO      CALC IF OVER
```

will transfer control to the instruction labeled CALC if the OVER flag is set to true. Otherwise, the instruction following the GOTO is executed.

Example:

```
GOTO      CALC IF NOT OVER
```

meaning control is transferred only if the OVER flag is set to false.

6.3 BRANCH

The BRANCH instruction transfers control to a statement specified by an index. The general form of the statement is as follows:

```
<label>  BRANCH    <index><prep><list>
```

where: <label> is an execution label (see section 2.).
<index> must be a numeric variable.
<prep> may be any valid preposition (see section 2.).
<list> is a list of execution labels separated by commas.

Programming Considerations:

-- The label is optional.

-- The condition flags are unchanged by the execution of this instruction.

-- The value of the index is unchanged by the execution of this instruction.

-- The index points to the label in the list where control is to be transfered.

-- If the index is n, then control is transfered to the nth label in the list. For example: if the index is 1, control is transfered to the first label in the list; if the index is 2, control is transfered to the second label in the list; and so on.

-- If the index is negative, zero, or larger than the number of labels in the list; then control continues in a sequential fashion.

-- If the index is a non-integer number, then only the digits preceding the decimal point are used while indexing into the list. For example: 1.50 is treated as if it were a 1, 1.99 is treated as if it were a 1, 2.00 is treated as if it were a 2, and 2.49 is treated as if it were a 2.

-- The list may be continued on the next line by using a colon in place of one of the commas.

Example:

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

If N = 1, then this BRANCH would be equivalent to a GOTO START. N = 2 would mean GOTO CALC while N = 3 would mean GOTO POINT.

6.4 CALL

The CALL instruction causes a subroutine to be executed after saving a pointer to the instruction immediately following the CALL instruction. When the subroutine is finished executing, it may then use the pointer that was saved to continue execution where it left off (see section 6.5). Using subroutines allows the same group of statements to be executed at many places in the user's program, simply by CALLing the subroutine. The format of the statement may be one of the following:

```
1)   <label1> CALL      <label2>
2)   <label1> CALL      <label2> IF <flag>
3)   <label1> CALL      <label2> IF NOT <flag>
```

where: <label1> is an execution label (see section 2.).
<label2> is an execution label.
<flag> is one of the condition flags (see section 6.1).

Programming Considerations:

-- <label1> is optional.

-- <label2> must be a label on the first instruction of the subroutine to be executed.

-- The condition flags are unchanged by the execution of this statement.

-- The return address (the pointer to the instruction immediately following the CALL statement) is saved by pushing it onto the subroutine call stack.

-- The subroutine call stack is eight levels deep. This means that, unless an entry is cleared from the stack (typically by a RETURN instruction), a stack overflow error will occur when the ninth CALL instruction is executed.

-- Note that if a page swap is invoked by the subroutine CALL, then CALLing the subroutine is considerably more time consuming than executing the code in line. The space used for Databus programs is virtual in nature to allow very large programs. This means that pages of the user's program must be swapped in and out of memory. If a subroutine happens to be on a different page than a CALL to that subroutine, then a page swap may become necessary. Therefore, in many cases it can be better to put 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.

-- Execution of a CHAIN statement will clear the subroutine call stack.

-- A CALL statement with format (2) will call the subroutine only if the specified condition flag is set to true; otherwise, program control continues in a sequential fashion.

-- A CALL statement with format (3) will call the subroutine only if the specified condition flag is set to false.

Example:

```
CALL      FORMAT
```

will execute the subroutine FORMAT.

Example:

```
CALL      XCOMP IF LESS
```

will execute the subroutine XCOMP if the LESS flag is set to true.

6.5 RETURN

The RETURN instruction is used to return from a subroutine when execution of that subroutine is completed. This statement may have one of the following formats:

1) <label> RETURN
2) <label> RETURN IF <flag>
3) <label> RETURN IF NOT <flag>

where: <label> is an execution label (see section 2.).
<flag> is a condition flag (see section 6.1).

Programming Considerations:

-- <label> is optional.

-- Control is returned to the instruction pointed to by the top element on the subroutine call stack.

-- The condition flags are unchanged by the execution of this statement.

-- A RETURN with format (2) will return control only if the specified condition flag is set to true; otherwise, program control continues in a sequential fashion.

-- A RETURN with format (3) will return control only if the specified condition flag is set to false.

Example:

```
RETURN
```

will transfer control to the instruction pointed to by the top element of the subroutine call stack.

Example:

```
RETURN IF ZERO
```

will transfer control to the instruction pointed to by the top element of the subroutine call stack only if the ZERO flag is set to true.

6.6 ACALL

The ACALL instruction is used to invoke an Assembler language routine. The individual interpreter manual should be consulted for the particular implementation. The format of the instruction is:

```
<label>  ACALL     <svar><prep><nslist>
```

where: <label> is an execution label.
<svar> is a string variable.
<prep> is a preposition.
<nslist>is a list of numeric or character string variables separated by a comma (,). The list may be continued on another line by placing a colon (:) after the last variable on the line to be continued. These variables are available to the Assembler routine.

Programming Considerations:

-- <label> is optional.

-- <svar> may be any string variable defined in the user's program. This variable is used by the interpreter before execution of the user's Assembler routine takes place.

-- <nslist> is optional.

-- <nslist> must consist of character string or numeric variables.

Example:

```
A       DIM      10
```

```
B       INIT    "12345"
C       FORM    "6.725"
        ACALL   A,B,C
```

6.7 STOP

The STOP instruction is the normal manner of terminating the execution of a Databus program. See the manual on the interpreter that you are using for more details on the action taken when a STOP is executed. Typically, executing a STOP instruction is equivalent to executing a CHAIN to the MASTER program for the port executing the STOP. This statement may have one of the following formats:

1) <label> STOP
2) <label> STOP IF <flag>
3) <label> STOP IF NOT <flag>

where: <label> is an execution label (see section 2.).
<flag> is a condition flag (see section 6.1).

Programming Considerations:

-- <label> is optional.

-- Typically executing a STOP is equivalent to executing a CHAIN to the MASTER program for the port executing the STOP.

-- See the manual on the interpreter executing the STOP instruction for the details on the action taken when the STOP is executed.

-- A STOP with format (2) will terminate only if the specified condition flag is set to true; otherwise, program control continues in a sequential fashion.

-- A STOP with format (3) will terminate only if the specified condition flag is set to false.

Example:

```
        STOP
```

will cause program execution to terminate normally.

Example:

will cause program execution to terminate normally only if the ZERO flag is set to false. Note that EQUAL is just another name for the ZERO flag.

6.8 CHAIN

This instruction is used to cause a Databus program (other than the one currently being executed) to be loaded and executed. One of the following general formats may be used:

1) <label> CHAIN <slit>
2) <label> CHAIN <svar>

where: <label> is an execution label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<svar> is a string variable (see section 4.2).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- The value of <svar> is unchanged by the execution of this instruction.

-- Only those DOS files that have an extension of /DBC can be loaded and executed.

-- Control is passed to the first executable statement of the program that is to be loaded and executed.

-- The string literal, when using format (1), specifies the DOS name of the Databus program to be executed.

-- The string variable, when using format (2), specifies the DOS name of the Databus program to be executed.

-- The extension is not furnished by the string literal or string variable. (/DBC is assumed as the extension.)

-- One of the following rules is used to build the DOS name from the string in the string variable or string literal:

a) The characters used start with the formpointed character and continue until eight characters have been obtained, or

b) if the logical end of string is reached before eight characters have been obtained, the remainder of the eight characters are assumed to be blanks.

-- The character used to specify the drive number is obtained from the string variable or string literal using one of the following rules:

a) If (a) above is used to obtain the name, then the character after the eighth character is used as the drive specification, or

b) If (b) above is used to obtain the name, then the character following the one pointed to by the logical length pointer is used as the drive specification, or

c) If the last character obtained from the string is physically the last character in the string, then the drive number is unspecified.

-- If the character used as the drive specification is not an ASCII digit (0 through 9), then all drives will be searched for the file (starting with drive 0 and ending with the highest numbered drive that is on-line).

-- If the drive number is unspecified, all drives will be searched for the file (starting with drive 0 and ending with the highest numbered drive that is on-line).

-- If the character used as the drive specification is an ASCII digit, then only the drive with that number will be searched to find the file.

-- Shift key inversion is enabled when a CHAIN instruction is executed (see section 9.1.3.15).

-- The trap locations are cleared after a CHAIN instruction is executed (see section 6.9).

-- The condition flags are all set to false by the execution of this statement.

-- All logical files that are open when a CHAIN instruction is executed, are closed without space deallocation (see section 12.3.2). Closing the files does not automatically write an

end-of-file mark.

-- The subroutine call stack is cleared by the execution of this statement (see section 6.4).

Assume that the following statement is used to define NXTPRGM for all of the following examples:

```
NXTPRGM  INIT      "PAYROLL11"
```

Example:

```
        RESET     NXTPRGM TO 9          SET THE LOGICAL
.                                       LENGTH POINTER TO 9
.
        RESET     NXTPRGM TO 4          SET THE FORMPOINTER TO 4
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named ROLL11/DBC from any drive on which it can be found.

Example:

```
        RESET     NXTPRGM TO 8          SET THE LOGICAL
        LENSET    NXTPRGM               LENGTH POINTER TO 8
.
        RESET     NXTPRGM TO 4          SET THE FORMPOINTER TO 4
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named ROLL1/DBC from drive 1.

Example:

```
        RESET     NXTPRGM TO 8          SET THE LOGICAL
        LENSET    NXTPRGM               LENGTH POINTER TO 8
.
        RESET     NXTPRGM TO 1          SET THE FORMPOINTER TO 1
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named PAYROLL1/DBC from drive 1.

Example:

```
        RESET     NXTPRGM TO 9          SET THE LOGICAL
.                                       LENGTH POINTER TO 9
.
        RESET     NXTPRGM TO 1          SET THE FORMPOINTER TO 1
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named PAYROLL1/DBC from drive 1.

Example:

```
        RESET     NXTPRGM TO 7          SET THE LOGICAL
        LENSET    NXTPRGM               LENGTH POINTER TO 7
.
        RESET     NXTPRGM TO 1          SET THE FORMPOINTER TO 1
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named PAYROLL/DBC from drive 1.

Example:

```
        RESET     NXTPRGM TO 3          SET THE LOGICAL
        LENSET    NXTPRGM               LENGTH POINTER TO 3
.
        RESET     NXTPRGM TO 1          SET THE FORMPOINTER TO 1
        CHAIN     NXTPRGM
```

this CHAIN instruction will try to load and execute a program named PAY/DBC from any drive on which it can be found.

6.9 TRAP

TRAP is a unique instruction; because rather than taking action at the time it is executed, it specifies a transfer location for an event which may or may not occur during later execution. This statement has the following general format:

<label1> TRAP <label2> IF <event>

where: <label1> is an execution label (see section 2.).
<label2> is an execution label.
<event> is one of the following: PARITY, RANGE, FORMAT, CFAIL or IO.

Programming Considerations:

-- <label1> is optional.

-- <label2> must be the label on the statement where control will be transfered if the specified event occurs.

-- The condition flags are unchanged by the execution of this instruction.

-- The following trapable events may occur:

a) PARITY - this event is caused by a disk CRC error during a READ (see section 12.3.3) or the verification phase of a WRITE (see section 12.3.4). DOS retries several times to get a good CRC before causing this event.

b) RANGE - this event occurs when a record number is out of range. Typically this occurs when an attempt is made to read a record that has never been written. The DOS RANGE and FORMAT traps will cause a Databus RANGE trap.

c) FORMAT - this event occurs when an attempt is made to read non-numeric data into a numeric variable. The read stops at the list item in error so that the rest of the list items will not be changed. Note that this FORMAT trap is not the same as the DOS FORMAT trap.

d) CFAIL - this event occurs when an attempt to CHAIN to another program cannot be completed or when an attempt to execute a ROLLOUT cannot be completed. Typically this occurs when attempting to CHAIN to a program that does not exist.

e) IO - this event occurs when a disk I/O error occurs (for more details about these I/O errors, see the user's guide of the appropriate interpreter). Typically this trap is used only for detecting whether a file exists or not. It is a good idea to keep this trap clear whenever it is not being used specifically to detect the presence of a file. This will prevent confusion if one of the other conditions occurs.

Example:

```
TRAP      PREP IF IO
OPEN      FILE,"DATA"
```

```
            GOTO        NSI
PREP        PREPARE     FILE,"DATA"
            RETURN
NSI         TRAPCLR     IO
```

-- The only action taken at the time that the TRAP instruction is executed is to save a pointer to the statement with <label2>. <event> specifies which trap.

-- Any traps that have been set, remain set until they are cleared.

-- If an event occurs and the trap is not set, the action taken depends upon the interpreter (see the user's guide for the interpreter you are using). Typically an error message is displayed and a CHAIN to that port's MASTER program occurs.

-- If an event occurs and the trap is set, then the action taken is as follows:

   a) The control transfer is equivalent to executing a
      ```
      CALL        <label2>
      ```
      instruction.

   b) This pseudo-CALL statement is executed as if it had been inserted immediately after the statement which caused the event to occur.

-- Whenever a certain event is trapped, the trap for that event is cleared. This means that, if the event is to be trapped again, another TRAP instruction will have to be executed to reset the trap.

-- Note that all of the traps are cleared whenever a CHAIN occurs. Therefore, each program must initialize all of the traps it wishes to use.

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.

6.10 TRAPCLR

This instruction will clear the specified trap. This statement has the following general format:

<label> TRAPCLR <event>

where: <label> is an execution label (see section 2.).
<event> is one of the following: PARITY, RANGE, FORMAT, CFAIL or IO. (For an explanation of each of the events, see section 6.9.)

Programming Considerations:

-- <label> is optional.

-- The condition flags are unchanged by the execution of this instruction.

Example:

```
TRAPCLR   PARITY
```

will clear the parity trap previously set.

6.11 ROLLOUT

The ROLLOUT feature allows the execution of all programs to be temporarily suspended while a DOS command line is executed. This instruction is particularly useful when 1) a file needs to be sorted using the DOS SORT utility, 2) an index file needs to be created using the DOS INDEX utility, or 3) a file needs to be re-indexed using the DOS INDEX utility. This statement may have one of the following formats:

1) <label> ROLLOUT <svar>
2) <label> ROLLOUT <slit>

where: <label> is an execution label (see section 2.).
<svar> is a string variable (see section 4.2).
<slit> is a literal of the form "<string>" (see section 2.5).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- The value of <svar> is unchanged by the execution of this instruction.

-- The string variable, when using format (1), specifies the DOS command line to be executed.

-- The string literal, when using format (2), specifies the DOS command line to be executed.

-- Since there are some minor differences in the way the ROLLOUT instruction is executed, the user should consult the user's guide of the interpreter he is using.

-- The characters used to build the DOS command line are taken one at a time from the string; from the first character to the last character, as defined below.

   a) The first character of the DOS command line is the formpointed character.

   b) The last character of the DOS command line precedes the first occurance of one of the following characters:

      1. a character with a value less than 040 (octal), or

      2. the vertical bar character (0174 octal), or

      3. a character with its sign bit set. The physical end-of-string character, 0203 (octal), fits into this category.

   In the normal case, this means the string used will be that from under the formpointer up through the physical end of the string. To use a string that is shorter than the physical length of the variable, a vertical bar should be stored in the appropriate position.

-- A CFAIL trap will occur if the string variable is null.

-- See the user's guide of the appropriate interpreter for other causes of CFAIL traps when attempting a ROLLOUT.

-- When the ROLLOUT instruction is executed the following actions are taken:

   a) Everything necessary to restore the interpreter to its

previous state is saved on disk.

b) DOS is then brought up at the console.

c) The operator at the console loses the information that was on the screen at the time of the ROLLOUT.

d) The DOS command line (obtained from the string variable or literal) is then supplied to the DOS command interpreter exactly as if it had been keyed in from the console.

-- To return the interpreter to the state it was in previous to the ROLLOUT, the interpreter's rollout return program should be executed. (For more details about the rollout return program, see the user's guide of the appropriate interpreter.) In the remainder of this manual the rollout return program will be refered to as DSBACK/CMD, or more simply DSBACK.

-- To execute the rollout return program, the name of the DSBACK command should be entered as a DOS command line. Generally this will cause the following actions:

a) DSBACK re-initializes the console screen. This does <u>not</u> return the screen to the display condition it was in before the ROLLOUT. That screen image is lost.

b) The information that was saved on disk by the ROLLOUT is then used to restore the interpreter to its previous state.

c) All ports are returned to their previous point of execution when the ROLLOUT occurred.

d) Execution of the program that caused the ROLLOUT is continued with the instruction following the ROLLOUT instruction.

-- The condition flags are restored by DSBACK.

-- The execution of a ROLLOUT may be very inconvenient to the users at other ports since execution of their programs will be suspended for an indefinite period of time. Unless told that a ROLLOUT has occured, users at the other ports will not know what is happening. 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.

-- The system clock is restored to the value it had before the ROLLOUT occured. This means that every time a ROLLOUT occurs the clock will lose time. In those environments where it is necessary for the system clock to be accurate, the rollout return program which includes time and date initialization should be used instead of DSBACK. In the remainder of this manual the rollout return program which includes time and date initialization will be refered to as DSBACKTD/CMD or more simply DSBACKTD (for more details see the user's guide of the appropriate interpreter). Note that, DSBACKTD functions the same as DSBACK with the exception that the new time and date are requested before restoring the interpreter. This rollout return program requires the operator to be at the console to enter the time and date.

-- ** WARNING ** The operations that were taking place under the interpreter must not be modified in any way. One of the items saved on disk when a ROLLOUT occurs is an image of the DOS file structure as it was under the interpreter. If the DOS file structure is changed by a program executing under DOS, then the image saved on disk may not be accurate any longer. If this image is no longer accurate when the interpreter is restored, terrible things may happen to the DOS file structure as well as the interpreter system. Some precautions that should be considered while executing under DOS are listed below.

a) The MASTER and ANSWER programs must not be re-compiled.

b) Any file that is open at the time when a ROLLOUT occured must not be modified or deleted.

c) The object code of any program that was executing when the ROLLOUT occured must not be changed.

d) The disks that contain any files in use by the interpreter must not be moved to another disk drive.

e) The disks that contain any files in use by the interpreter must not be removed from the disk drive.

Other operators using a Datashare system should be notified when a ROLLOUT is about to occur. This courtesy will prevent frustration when the other operators begin getting no response.

-- Rolling out to the configuration program (for details see the appropriate interpreter manual) has no effect on the system

configuration when DSBACK is used to restart the interpreter.

Example:

Assume that a Databus program has built two files, AFILE/TXT and CFILE/TXT. Also, assume that these files need to be sorted.

This can be accomplished by building the following file named ROLCHAIN/TXT.

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

then executing the following instruction.

```
ROLLOUT   "CHAIN ROLCHAIN"
```

This would cause execution of the interpreter to be suspended, and the following DOS command to be executed (for more details on the DOS CHAIN command, see the DOS user's guide).

```
CHAIN ROLCHAIN
```

Executing this command would then cause the commands in the file ROLCHAIN/TXT to be executed one after another. First, the file AFILE/TXT would be sorted and and then written into file BFILE/TXT. Second, the file CFILE/TXT would be sorted and then written into file DFILE/TXT. And last, the DSBACK command would be executed to cause execution of the interpreter to be continued.

Note that if DSBACK had not been included in the chain file the operator would have had to restore the system. Also note that if, for any reason, the chain file did not go to completion; then the operator would have had to execute the DSBACK command from the console.

6.12 PI

This instruction (Prevent Interruptions) enables the programmer to prevent his background program from being interrupted for up to 20 Databus instruction executions. It is particularly useful in preventing any other port from modifying a disk record while that record is in the process of being updated (see appendix D). This instruction has the following general format:

```
<label>  PI        <dnum>
```

where: <label> is an execution label (see section 2.).
<dnum> is a decimal number.

Programming Considerations:

-- <label> is optional.

-- <dnum> must be between 1 and 20, inclusive.

-- <dnum> specifies the number of Databus instructions to be executed before allowing an interuption. The PI instruction is not included as one of these instructions.

-- The PI instruction may be used to postpone any of the following background interuptions:

   a) the keyboard interruption procedure (see section 9.1.5.3),

   b) by a higher priority execution being requested on another port (caused by the termination of a foreground process), or

   c) by the port using up its share of the background time.

-- This instruction has no effect upon the hardware one millisecond interrupt used to perform all port and printer I/O.

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

-- If interrupts are prevented; the execution of any instruction that causes background to wait for I/O to finish will cancel the effect of the PI instruction. DISPLAY, KEYIN, CONSOLE and PRINT are examples of instructions that cause background to wait for I/O to finish.

-- If a PI instruction is executed while interruptions are already prevented, execution of that program is aborted. This prevents a program from being able to prevent interruptions for more than 20 instruction executions.

-- Note that the PI instruction can only prevent those interrupts that are under control of the interpreter. The PI instruction cannot be used to prevent interruptions such as power failures

or the system operator restarting the processor. This means that when designing complex data file structures, the programmer should take care that any interruptions will do as little harm as possible. The PI instruction is primarily useful in preventing interruptions (such as the typist bumping the interupt key or a different port modifying the critical record) while modifying records that are critical to maintaining the file structure.

Example:

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

Interruptions will be prevented from the PI instruction through the UPDATE instruction. Note that no other Datashare port can modify the record being updated until this port has completed its modification of the record. Using this technique, more than one port can reference the "QuanTitY ON Hand" and receive an up-to-date answer.

6.13 TABPAGE

This instruction is used to force sections of a program to begin at the first of an object code page. Execution speed can be enhanced in this way by reducing object code page accesses. This instruction has the following general format:

```
<label>  TABPAGE
```

where: <label> is an execution label (see section 2.).

Programming Considerations:

-- <label> is optional.

-- A page of object code is 250 bytes long. Page boundaries can be detected in the listing of a program by looking at the three least significant digits of the location counter and noting one of the following:

   a) a location counter change from 772 (octal) to 001 (octal), or

b) a location counter change from 372 (octal) to 401 (octal).

-- Compilation of a TABPAGE instruction forces the instruction following the TABPAGE to be put at the first of the next page of object code.

-- Execution of a TABPAGE instruction causes control to be transferred to the first byte of the next page.

-- Note that liberally scattering TABPAGE instructions throughout a user program will in general not result in an increase in execution speed. Instead, the usual effect is to increase the rate of thrashing of the program.

-- TABPAGE is best used to force tight loops to reside entirely within one or two pages.

6.14 DSCNCT

The DSCNCT instruction is the normal method for a program to terminate when executing as a remote slave port. This instruction has the following general format:

<label> DSCNCT

where: <label> is an execution label (see section 2.).

Programming Considerations:

-- <label> is optional.

-- The DSCNCT instruction causes the following actions:

a) All telephone communication activities are terminated.

b) The telephone is hung up.

c) The remote station is returned to DOS.

d) The equivalent of a CHAIN to the ANSWER program is executed.

CHAPTER 7. CHARACTER STRING HANDLING INSTRUCTIONS

The character string handling instructions are used to change the contents of character strings, or the string attributes (Logical Length, Formpointer). Generally all string handling instructions have the following form:

<label> <oper> <soper><prep><doper>

where: <label> is an execution label.
<oper> is the string operation.
<soper> is the source operand.
<prep> is a preposition.
<doper> is the destination operand. .

The reader should be familiar with the various DATABUS data types. This information is contained in chapter 4. and should be read before continuing.

7.1 MOVE

The MOVE instruction transfers the contents of the source string into the destination string. Four (4) different types of move operations are defined:

1) MOVE character string to character string.
2) MOVE character string to numeric string.
3) MOVE numeric string to character string.
4) MOVE numeric string to numeric string.

The first three (3) MOVE operations will be discussed in this chapter, the fourth type will be discussed in Chapter 8 on Arithmetic Instructions.

7.1.1 MOVE (character string to character string)

This MOVE instruction transfers the contents of the source operand into the destination operand. This instruction has the following formats:

1) <label> MOVE <ssvar><prep><dsvar>
2) <label> MOVE <slit><prep><dsvar>

where: <label> is an execution label.
<ssvar> is the source string variable.
<prep> is a preposition.
<dsvar> is the destination string variable.
<slit> is the source string literal.

Programming Considerations:

-- <label> is optional.

-- 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. When transfer is complete, the formpointer of the destination string 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. Transfer stops with the character that would have overstored the ETX.

-- A null source string (formpointer=0) will cause:

a. the destination variable formpointer to be set to zero.

b. no characters are moved.

c. the length pointer of the destination variable is not changed.

Example:

```
VAR        LL FP  Contents

STRING1    6  1   ABCDEF        ETX
STRING2    6  1   DOGCAT        ETX

           MOVE STRING1 TO STRING2

The following variable(s) will be changed:
STRING2    6  1   ABCDEF        ETX
The following flag(s) will be set: None
```

Example:

```
STRING1   4  2   ABCDXLM       ETX
STRING2   6  3   DOGCAT        ETX

          MOVE STRING1 TO STRING2

The following variable(s) will be changed:
STRING2   3  1   BCDCAT        ETX
The following flag(s) will be set: None
```

Example:

```
STRING1   4  2   ABCDXLM       ETX
STRING2   6  3   DOGCAT        ETX

          MOVE "HELLO" TO STRING2

The following variable(s) will be changed:
STRING2   5  1   HELLOT        ETX
The following flag(s) will be set: None
```

Example:

```
STRING1   7  2   ABCDEFG       ETX
STRING2   4  3   HIJKL         ETX

          MOVE STRING1 TO STRING2

The following variable(s) will be changed:
STRING2   5  1   BCDEF         ETX
The following flag(s) will be set: EOS
```

Example:

```
STRING1   7  0   ABCDEFG       ETX
STRING2   4  3   HIJKL         ETX

          MOVE STRING1 TO STRING2

The following variable(s) will be changed:
STRING2   4  0   HIJKL         ETX
The following flag(s) will be set: None
```

7.1.2 MOVE (character string to numeric string)

This MOVE transfers the contents of the source character string to the destination numeric string. The instruction has the following formats:

```
1)   <label>  MOVE      <ssvar><prep><dnvar>
2)   <label>  MOVE      <sslit><prep><dnvar>
```

where: <label> is an execution label.
<ssvar> is the source string variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<sslit> is the source string literal.

Programming Considerations:

-- <label> is optional.

-- A character string will be moved to a numeric string only if the portion of the character string from the formpointer through the logical length is of valid numeric format (at most one decimal point, sign, and digits only).

-- The transfer from the source string starts at the formpointer and proceeds through the logical length of the source string.

-- The source character string will be reformatted and rounded to fit the destination numeric string.

-- If any of the most significant digits or sign is lost in the process of truncation, then the OVER flag is set and the destination numeric variable is not changed.

-- A null source string (formpointer=0) will result in the destination variable not being changed.

Example:

```
VAR        LL FP  Contents

STRING     9  3   AB100.327      ETX
NUMBER     0200   _39.00         ETX

           MOVE STRING TO NUMBER

The following variable(s) will be changed:
NUMBER     0200   100.33         ETX
```

The following flag(s) will be set: None

Example:

```
STRING1    9  3   AB10X.327      ETX
NUMBER     0200   _39.00         ETX

           MOVE STRING1 TO NUMBER
```

The following variable(s) will be changed: None
The following flags will be set: None

Example:

```
NUMBER     0200   12345.3        ETX

           MOVE "935" INTO NUMBER
```

The following variable(s) will be changed:
NUMBER 0200 __935.0 ETX
The following flag(s) will be set: None

Example:

```
STRING     5  0   ABCDE          ETX
NUMBER     0200   __935.0        ETX

           MOVE STRING TO NUMBER
```

The following variables will be changed· None
The following flag(s) will be set: None

7.1.3 MOVE (numeric string to character string)

This instruction transfers the contents of the source numeric string to the destination character string. The instruction has the following formats:

1) <label> MOVE <snvar><prep><dsvar>
2) <label> MOVE <nlit><prep><dsvar>

where: <snvar> is the source numeric variable
<prep> is a preposition.
<dsvar> is the destination character string variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- Transfer from the source numeric string starts with the first character of the string and continues until the source numeric ETX is reached or until the ETX of the destination string is about to be overstored.

-- Transfer into the destination character string begins with the first physical character and continues until either the source string ETX is encountered or the destination character string ETX is about to be overstored.

-- The formpointer is set to one (1) and the logical length is set to point to the last character transferred into the destination string.

-- The EOS flag is set if the ETX would have been overstored in the destination character string. The transfer stops with the character before the one that would have overstored the ETX.

Example:

```
    VAR        LL FP  Contents

    NUMBER     0200   100.33         ETX
    STRING2    9  3   AB100.327      ETX

               MOVE NUMBER TO STRING2

    The following variable(s) will be changed:
    STRING2    6  1   100.33327      ETX
    The following flag(s) will be set: None
```

Example:

```
    NUMBER     0200   10.35789       ETX
    STRING2    5  3   ABCDE          ETX

               MOVE NUMBER TO STRING2

    The following variable(s) will be changed:
    STRING2    5  1   10.35          ETX
    The following flag(s) will be set: EOS
```

7.2 APPEND

APPEND appends the source string (character or numeric) to the destination string. The instruction has the following formats:

```
1)   <label>  APPEND    <ssvar><prep><dsvar>
2)   <label>  APPEND    <snvar><prep><dsvar>
3)   <label>  APPEND    <slit><prep><dsvar>
```

where: <label> is an execution label.
<ssvar> is the source string variable.
<prep> is a preposition.
<dsvar> is the destination string variable.
<snvar> is the source numeric variable.
<slit> is the source string literal.

Programming Considerations:

-- <label> is optional.

-- The portion of the source defined by one of the following:

1) For source character strings, the formpointed character through the logical length of the source character string.

2) For numeric strings, the first character through the physical end of string (ETX)

is appended to the destination character string.

-- The source string is appended starting after the formpointed character in the destination string.

-- The source string pointers are not changed.

-- The destination string formpointer and logical length point to the last character transferred.

-- The EOS flag is set if the portion of the source string that is to be moved cannot be contained in the destination string. All of the characters that will fit, will be appended.

Example:

```
VAR        LL FP  Contents
```

```
STRING1   8  6   JOHN_DOE       ETX
STRING2   11 11  MARY_JONES_____________ETX

          APPEND STRING1 TO STRING2

The following variable(s) will be changed:
STRING2   14 14  MARY_JONES_DOE_________ETX
The following flag(s) will be set: None
```

Example:

```
STRING2   08 09  MARY_JONES_____________ETX

          APPEND ".XX.YY." TO STRING2

The following variable(s) will be changed:
STRING2   15 15  MARY_JONE.XX.YY._______ETX
The following flag(s) will be set: None
```

Example:

```
NUMBER    0200   100.33        ETX
STRING2   9  2   ABCDEFGHI     ETX

          APPEND NUMBER TO STRING2

The following variable(s) will be changed:
STRING2   8  8   AB100.33I     ETX
The following flag(s) will be set: None
```

7.3 MATCH

MATCH compares two character strings. The instruction has the following formats:

```
1)   <label>  MATCH     <svar><prep><svar>
2)   <label>  MATCH     <slit><prep><svar>
```

where: <label> is an execution label.
<svar> is a string variable.
<prep> is a preposition.
<slit> is a string literal.

Programming Considerations:

-- <label> is optional.

-- MATCH compares two character strings starting at the formpointer of each string and stopping when the end of either operand's string is reached.

-- The formpointers and length pointers of both strings are unchanged.

-- The length of each string is defined to be:

    length = logical length - formpointer + 1

-- If all of the characters that are compared match, then the EQUAL flag is set and the following computation is made:

        L = (length of destination string) -
        (length of source string)

    The LESS flag is set to indicate that L is negative.

-- If all of the characters that are compared do not match, then the following computation is made:

        D = (octal value of first non matching destination character) -
        (octal value of first non matching source character)

    The LESS flag is set if D is less than zero.

-- If either the source or destination string formpointer is zero before the operation, then the LESS and EQUAL flags are cleared and the EOS flag is set.

Example:

```
VAR        LL FP  Contents

STRING1    5  1   ABCDE          ETX
STRING2    4  1   ABCD           ETX

           MATCH STRING1 TO STRING2
```

The following flag(s) will be set: EQUAL, LESS

Example:

```
STRING1    3  1   ABC            ETX
STRING2    1  1   Z              ETX
```

```
            MATCH STRING1 TO STRING2
```

The following flag(s) will be set: None

Example:

```
STRING1   3  1   ZZZ          ETX
STRING2   3  1   AAA          ETX

            MATCH STRING1 TO STRING2
```

The following flag(s) will be set: LESS

Example:

```
STRING1   6  4   XXXABC       ETX
STRING2   5  3   YYABC        ETX

            MATCH STRING1 TO STRING2
```

The following flag(s) will be set: EQUAL

Example:

```
STRING2   5  1   ABCDE        ETX

            MATCH "ABCD" TO STRING2
```

The following flag(s) will be set: EQUAL

Example:

```
STRING2   5  0   ABCDE        ETX

            MATCH "ABCDE" TO STRING2
```

The following flag(s) will be set: EOS

7.4 CMOVE

The CMOVE instruction moves a character from the source operand into the destination character string. The instruction has the following formats:

```
1)   <label>  CMOVE     <ssvar><prep><dsvar>
2)   <label>  CMOVE     <char><prep><dsvar>
```

where: <label> is an execution label.
<ssvar> is the source string variable.
<prep> is a preposition.
<dsvar> is the destination string variable.
<char> is the one character source literal string.

Programming Considerations:

-- <label> is optional.

-- Transfer from the source string starts with the character under the formpointer.

-- Transfer into the destination string starts with the character under the formpointer.

-- Only one character is moved.

-- The source string logical length and formpointer are not modified.

-- If either variable has a formpointer of zero (0), then the EOS flag is set and no transfer occurs.

Example:

```
    VAR       LL FP  Contents

    STRING1   5  3   ABCDE          ETX
    STRING2   3  2   XXX            ETX

              CMOVE STRING1 TO STRING2

    The following variable(s) will be changed:
    STRING2   3  2   XCX            ETX
    The following flag(s) will be set: None
```

Example:

```
    STRING2   3  2   1234           ETX

              CMOVE "X" TO STRING2

    The following variable(s) will be changed:
    STRING2   3  2   1X34           ETX
    The following flag(s) are set: None
```

7.5 CMATCH

CMATCH compares a single character from the source string to a character in the destination string. The instruction has the following formats:

```
1)   <label>  CMATCH    <ssvar><prep><dsvar>
2)   <label>  CMATCH    <char><prep><dsvar>
3)   <label>  CMATCH    <ssvar><prep><char>
4)   <label>  CMATCH    <occ><prep><dsvar>
5)   <label>  CMATCH    <ssvar><prep><occ>
```

where:
- <label> is an execution label.
- <ssvar> is the source string variable.
- <prep> is preposition.
- <dsvar> is the destination string variable.
- <char> is a one character string literal.
- <occ> is an octal control character.

Programming Considerations:

-- <label> is optional.

-- The character compared from the source string is the character from under the formpointer.

-- The character compared from the destination string is the character from under the formpointer.

-- If the two characters match, then the EQUAL flag is set.

-- If the two characters do not match then the LESS flag is set if the following difference (D) is negative:

D = (octal value of destination character) - (octal value of source character).

-- If a literal or octal control character is used in the source string then that character is the one used for the CMATCH operation.

-- If either operand has a formpointer of zero (0), then the EOS flag is set.

Example:

```
VAR       LL FP  Contents

STRING1   5  3   ABCDE          ETX
STRING2   3  1   CX             ETX

          CMATCH STRING1 TO STRING2
```

The following flag(s) are set: EQUAL

Example:

```
STRING2   4  2   XACD           ETX

          CMATCH "B" TO STRING2
```

The following flag(s) are set: LESS

Example:

```
ST        8  0   ABCDEFGH       ETX

          CMATCH "Y" TO ST
```

The following flag(s) are set: EOS

7.6 BUMP

The BUMP instruction increments or decrements the formpointer of a variable. The instruction has the following formats:

1) <label> BUMP <svar>
2) <label> BUMP <svar><prep><dcon>

where: <label> is an execution label.
<svar> is a string variable.
<prep> is a preposition.
<dcon> is a signed decimal constant.

Programming Considerations:

-- <label> is optional.

-- <dcon> is added to the formpointer and the result becomes the new string variable formpointer if the new formpointer is valid. Note that a valid formpointer must be in the range (1 to n) where n is the logical length for the string.

-- If <dcon> is not specified, then the string variable's formpointer is incremented by one (1).

-- The EOS flag is set if the BUMP instruction would have caused an invalid formpointer. The formpointer is not changed in this case.

Example:

```
VAR        LL FP  Contents

CAT        5  2   ABCDE          ETX

           BUMP CAT

The following variable(s) will be changed:
CAT        5  3   ABCDE          ETX
The following flag(s) will be set: None
```

Example:

```
CAT        5  4   ABCDE          ETX

           BUMP CAT BY -2

The following variable(s) will be changed:
CAT        5  2   ABCDE          ETX
The following flag(s) will be set: None
```

Example:

```
CAT        5  3   ABCDE          ETX

           BUMP CAT BY 3

The following variable(s) will be changed:
CAT        5  3   ABCDE          ETX
The following flag(s) will be set: EOS
```

7.7 RESET

RESET changes the value of the formpointer of the destination string to the value indicated by the second operand. The instruction has the following formats:

```
1)   <label>  RESET    <dsvar><prep><dcon>
2)   <label>  RESET    <dsvar>
3)   <label>  RESET    <dsvar><prep><ssvar>
4)   <label>  RESET    <dsvar><prep><snvar>
```

where: <label> is an instruction label.
<dsvar> is the destination string variable.
<prep> is a preposition.
<dcon> is a decimal constant.
<ssvar> is the source string variable.
<snvar> is the source numeric variable.

Programming Considerations:

-- <label> is optional.

-- RESET changes the value of the formpointer of the destination string to the value indicated by the second operand. If the second operand is not specified the formpointer will be reset to one (1).

-- If the second operand is a quoted character the formpointer of the destination string is changed to the following:

FP = (OCTAL value of ASCII character) - 037

-- If the second operand is a character string the character under the formpointer is accessed. The formpointer of the destination string is changed to the following:

FP = (OCTAL value of ASCII character) - 037

-- If the second operand is a numeric string the number is used as the value for the new formpointer.

-- If the new formpointer is past the logical length of the first operand, the logical length will be set to the value of the new formpointer. Note that under no circumstances will the logical length or formpointer be set outside the physical structure of the string.

-- The EOS flag will be set when any change in the logical length

of the destination string occurs.

-- The RESET instruction is very useful in code conversions and hashing of character string values as well as large string manipulation.

Example:

```
VAR        LL FP  Contents

XDATA      5  3   ABCDEFGHIJ    ETX

           RESET XDATA

The following variable(s) will be changed:
XDATA      5  1   ABCDEFGHIJ    ETX
The following flag(s) will be set: None
```

Example:

```
XDATA      5  2   ABCDEFGHIJ    ETX

           RESET XDATA TO 4

The following variable(s) will be changed:
XDATA      5  4   ABCDEFGHIJ    ETX
The following flag(s) will be set: None
```

Example:

```
XDATA      10 2   ABCDEFGHIJ    ETX
NUMBER     0200   8             ETX

           RESET XDATA TO NUMBER

The following variable(s) will be changed:
XDATA      10 8   ABCDEFGHIJ    ETX
The following flag(s) will be set: None
```

Example:

```
XDATA      6  2   ABCDEFGHIJ    ETX
NUMBER     0200   8             ETX

           RESET XDATA TO NUMBER

The following variable(s) will be changed:
XDATA      8  8   ABCDEFGHIJ    ETX
```

The following flag(s) will be set: EOS

Example:

```
XDATA     10 8   1234567890    ETX
STRING    5  4   ABC!E         ETX

          RESET XDATA TO STRING

The following variable(s) will be changed:
XDATA     10 2   1234567890    ETX
Note: The ASCII value of a ! is a octal 041.
The following flag(s) are set: None
```

7.8 ENDSET

ENDSET causes the operand's formpointer to be changed to the value of the logical length. This instruction has the following format:

```
<label>  ENDSET    <dsvar>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

Example:

```
VAR       LL FP  Contents

CAT       10 4   1234567890    ETX

          ENDSET CAT

The following variable(s) will be changed:
CAT       10 10  1234567890    ETX
The following flag(s) will be set: None
```

Example:

```
DOG         6  4    1234567890     ETX

            ENDSET DOG

The following variable(s) will be changed:
DOG         6  6    1234567890     ETX
The following flag(s) will be set: None
```

7.9 LENSET

LENSET changes the operand's logical length to the value of the formpointer. The instruction has the following format:

```
        <label>  LENSET    <dsvar>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

Example:

```
VAR         LL FP  Contents

STRING      8  4   1234567890     ETX

            LENSET STRING

The following variable(s) will be changed:
STRING      4  4   1234567890     ETX
The following flag(s) will be set: None
```

Example:

```
XDATA       6  2   1234567890     ETX

            LENSET XDATA

The following variable(s) will be changed:
XDATA       2  2   1234567890     ETX
The following flag(s) will be set: None
```

7.10 CLEAR

CLEAR sets the logical length and formpointer of the operand to zero. This instruction has the following format:

```
<label>  CLEAR     <dsvar>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

Example:

```
VAR       LL FP  Contents

STRING    8  3   ABCDEFGHIJ    ETX

          CLEAR STRING
```

The following variable(s) will be changed:

```
STRING    0  0   ABCDEFGHIJ    ETX
```
The following flag(s) will be set: None

7.11 EXTEND

EXTEND increments the string variable's formpointer by one and stores a space into the new formpointed character. The logical length is set to the value of the new formpointer. This instruction has the following format:

```
<label>  EXTEND    <dsvar>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

-- The formpointer of the string variable is incremented by one. The logical length is set to the value of the new formpointer.

-- If the new formpointed character is the ETX, then the EOS flag is set and the formpointer and logical length are left as they were before the EXTEND instruction was executed.

Example:

```
    VAR        LL FP  Contents

    STRING     10 3   ABCDEFGHIJ    ETX

               EXTEND STRING

    The following variable(s) will be changed:
    STRING     4  4   ABC_EFGHIJ    ETX
    The following flag(s) will be set: None
```

Example:

```
    STRING     10 10  ABCDEFGHIJ    ETX

               EXTEND STRING

    The following variable(s) will be changed:
    STRING     10 10  ABCDEFGHIJ    ETX
    The following flag(s) will be set: EOS
```

7.12 LOAD

LOAD performs a MOVE from the selected character string (using an index for selection) to the destination character string. The instruction has the following formats:

```
<label>  LOAD      <dsvar><prep><index><prep><list>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.
<prep> is a preposition.
<index> is a numeric string used for selecting a string variable from the <list>.

<list> is a list of string variables.

The LOAD instruction to use when <list> is a set of numeric variables is covered in Chapter 8 on Arithmetic Instructions. This discussion deals only with the case when <list> is a set of string variables.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

-- <index> is a numeric variable. If this variable is not an integer, then the quantity is truncated and the integer portion used as the index for list selection.

-- If the <index> does not correspond to a variable in the <list>, then the LOAD instruction is simply ignored.

-- <list> must contain string variables only. The <list> may be continued if necessary by using the colon (:) instead of the comma (,) after the last variable used on the line to be continued.

-- This instruction works exactly like the MOVE instruction (character string to character string) after the variable has been selected from the list.

-- An <index> quantity of one (1) corresponds to the first variable in the <list> and an <index> quantity of n corresponds to the nth variable in the <list>.

Example:

```
    VAR       LL FP  Contents

    DESTIN    10 5   ABCDEFGHIJ     ETX
    INDEX     0200   __2.9          ETX
    S1        5  1   11111          ETX
    S2        5  2   22222          ETX
    S3        5  3   33333          ETX

              LOAD DESTIN FROM INDEX OF S1,S2:
                 S3

    The following variable(s) will be changed:
    DESTIN    4  1   2222EFGHIJ     ETX
```

The following flag(s) will be set: None

Example:

```
DESTIN     5  1   ABCDE          ETX
INDEX      0200   __3.7          ETX
S1         6  1   111111         ETX
S2         7  1   2222222        ETX
S3         8  1   33333333       ETX
S4         9  1   444444444      ETX

           LOAD DESTIN FROM INDEX OF S1,S2,S3,S4

The following variable(s) will be changed:
DESTIN     5  1   33333          ETX

The following flag(s) will be set: EOS
```

7.13 STORE

STORE selects a variable from a list (using an index for selection) and performs a MOVE operation from the source string operand to the selected destination string variable. The instruction has the following formats:

```
1)   <label>  STORE    <ssvar><prep><index><prep><list>
2)   <label>  STORE    <slit><prep><index><prep><list>
```

where:
- <label> is an execution label.
- <ssvar> is the source string variable.
- <prep> is a preposition.
- <index> is the numeric variable which specifies which variable from <list> is to be selected as the destination variable for the MOVE operation.
- <list> is a list of string variables.
- <slit> is a string literal.

Programming Considerations:

-- <label> is optional.

-- <list> is a list of string variables, separated by commas (,). The list may be continued on the following line by using a colon (:) instead of a comma (,) after the last variable on the line to be continued.

-- <index> must be a numeric variable. If the <index> is not an integer, it is truncated and the integer portion is used as the index for list selection.

-- If the <index> does not correspond to a variable in the <list>, then the STORE instruction is simply ignored.

-- An <index> quantity of one (1) corresponds to the first variable in the <list> and an <index> quantity of n corresponds to the nth variable in the <list>.

-- All of the rules of the MOVE instruction apply after the list selection has been performed.

Example:

```
VAR       LL FP  Contents

SOURCE    8  5   12345678      ETX
I         0200   __2.3         ETX
D1        5  2   11111         ETX
D2        6  3   222222        ETX
D3        7  4   3333333       ETX

          STORE SOURCE INTO I OF D1,D2:
             D3

The following variable(s) will be changed:
D2        4  1   567822        ETX
The following flag(s) will be set: None
```

Example:

```
IND       0200   3             ETX
D1        5  1   12345         ETX
D2        4  2   ABCD          ETX

          STORE "890" INTO IND OF D1,D2
```

The instruction would not be executed because the index is out of range.

7.14 CLOCK

CLOCK allows a DATABUS program access to the interpreter's time clock, day, and year. The instruction has the following formats:

<label> CLOCK <item><prep><svar>

where: <label> is an execution label.
<item> may be one of the following:

1) TIME to access the time of day clock.
2) DAY to access the day of the year.
3) YEAR to access the year.

<prep> is one of the prepositions <,> or <TO>.
<svar> is a string variable that is to receive the requested information.

Programming Considerations:

-- <label> is optional.

-- Only the prepositions <,> and <TO> are valid.

-- <svar> must be a string variable.

-- The time clock (TIME) has the following format:

hh:mm:ss

where:

hh = hours tens and units digits with range (00 to 23).
mm = minutes tens and units digits with range (00 to 59)
ss = seconds tens and units digits with range (00 to 59).

-- The day of the year (DAY) has the following format:

ddd representing the hundreds, tens, and units digits of the day of year with range (001 to 366). The day expressed in this form is commonly termed the "Julian" day.

-- The year (YEAR) has the following format:

yy representing the tens and units of the year with range (00 to 99).

-- The CLOCK instruction simply performs a MOVE operation on information requested into the destination string variable.

-- The DATABUS programmer must be careful when using the CLOCK instruction to avoid getting erroneous results. When obtaining both the TIME and DAY, the program should first access the DAY and then the TIME. The program should then access the DAY again and insure that the DAY has not changed. If the DAY has changed, then the process should be repeated. When this procedure is followed, then the TIME and DAY correspond to each other.

-- The TIME, DAY, and YEAR are placed into the interpreter when the system is brought up. The CLOCK items are kept updated while the interpreter is running and are available to DATABUS programs.

-- The TIME is accurate to approximately 0.005 percent or five (5) seconds per day.

```
VAR        LL FP  Contents

TIME       8  2   XXXXXXXX       ETX
DAY        3  3   YYY            ETX
TEMP       3  2   ZZZ            ETX
YEAR       2  2   ZZ             ETX

           CLOCK DAY TO DAY
           CLOCK TIME TO TIME
           CLOCK YEAR TO YEAR
           CLOCK DAY TO TEMP
           MATCH DAY TO TEMP
           GOTO TIMEOK IF EQUAL
           CLOCK DAY TO DAY
           CLOCK TIME TO TIME
TIMEOK     ........

The following variable(s) will be changed:
TIME       8  1   13:10:52       ETX
DAY        3  1   134            ETX
YEAR       2  1   76             ETX
TEMP       3  1   134            ETX
```

The above would be correct if the time was 13 hours, 10 minutes, 15 seconds, the day of the year was the 134th, and the

year number was 76.

7.15 TYPE

This instruction checks the format of a character string variable for valid numeric string format. This instruction has the following format:

```
<label>  TYPE      <dsvar>
```

where: <label> is an execution label.
<dsvar> is the destination string variable.

Programming Considerations:

-- <label> is optional.

-- <dsvar> must be a string variable.

-- Only the logical string of <dsvar> is checked for valid numeric format (see section 4.1).

-- The EQUAL flag is set to true only when the logical string is a valid numeric string.

-- A null logical string is not a valid numeric string.

7.16 SEARCH

SEARCH compares a variable <key> to a list of variables <list> and yields an index <index> which indicates which variable in the <list> matched. The instruction has the following format:

```
<label>  SEARCH    <key><prep><blist><prep><nlist><prep><index>
```

where: <label> is an execution label.
<key> is the key variable.
<prep> is a preposition.
<blist> is the first variable in a list of contiguous variables.
<nlist> is a numeric variable which specifies the number of variables in the list to be searched.
<index> is a numeric variable produced by the interpreter which specifies which variable in the list (the

first of which was <blist>) matched the <key>.

Programming Considerations:

-- <label> is optional.

-- <key> and the variables in the list (the first of which is <blist>) should be of the same data type, either both string variables or both numeric variables.

-- <blist> is the name of the first variable in the list of contiguous variables to be used.

-- <nlist> is a numeric variable which specifies the number of variables in the list (the first of which is <blist>).

-- The logical string of <key> is compared to the logical string of a variable from the list (of which <blist> is the first). If the logical string length of <key> is less than the logical string length of the variable being compared (from the list), the match stops when the <key> logical string is exhausted. Therefore, it is not possible to get a match on a <key> variable whose logical string is longer than the logical string of the list variable.

-- The logical string lengths of the variables in the list may be different.

-- If the variable <nlist> is larger than the number of variables in the list, the search proceeds until the count <nlist> is exhausted.

-- <index> contains a one (1) if the first variable in the list matched <key>. A value of n for <index> indicates the nth variable in the list matched <key>. The EQUAL flag is also set if a match is found.

-- If none of the list variables matched <key> then a value of zero (0) is returned for <index> and the OVER flag is set.

Example:

VAR	LL	FP	Contents	
KEY	5	3	ABCDE	ETX
VAR1	8	1	12345678	ETX
VAR2	6	2	XCDE12	ETX

```
VAR3        4  3    FGHI            ETX
NVAR        0200    03              ETX
INDEX       0200    00              ETX

            SEARCH KEY IN VAR1 TO NVAR WITH INDEX

The following variable(s) will be changed:
INDEX       0200    _2              ETX
The following flag(s) will be set: EQUAL
```

Example:

```
KEY         5  3    ABCDE           ETX
V1          5  1    XXXXX           ETX
V2          3  1    YYY             ETX
V3          4  1    ZZZZ            ETX
N           0200    _3              ETX
I           0200    99              ETX

            SEARCH KEY IN V1 TO N USING I

The following variables will be changed:
I           0200    _0              ETX
The following flag(s) will be set: OVER
```

7.17 REPLACE

REPLACE (the compiler will also accept a mnemonic of REP) allows replacement of an ASCII character in a string variable by another ASCII character. The instruction has the following formats:

```
1)   <label>  REPLACE   <ssvar><prep><dsvar>
2)   <label>  REP       <ssvar><prep><dsvar>
3)   <label>  REPLACE   <slit><prep><dsvar>
4)   <label>  REP       <slit><prep><dsvar>
```

where: <label> is an execution label.
<ssvar> is the source string variable.
<prep> is a preposition.
<dsvar> is the destination string variable.
<slit> is a source string literal.

Programming Considerations:

-- <label> is optional.

-- The logical string of the source variable <ssvar> or literal <sslit> must contain pairs of characters defined as follows:

1) The first character of the pair is the character to be replaced in the destination string.

2) The second character of the pair is the character that is to replace the first of the pair wherever it appears in the destination string.

-- The source string is not modified.

-- The destination variable logical string is modified.

-- The EOS flag is set if the logical string length of the source operand is not even.

Example:

```
    VAR        LL FP  Contents

    DVAR       10 1   ABCDABCDAB     ETX
    ABVAR      4  1   AXDY           ETX

               REPLACE ABVAR IN DVAR

    The following variable(s) will be changed:
    DVAR       10 1   XBCYXBCYXB     ETX
    The following flag(s) will be set: None
```

Example:

```
    DVAR       10 5   ABCDABCDAB     ETX
    ABVAR      4  3   AXDY           ETX

               REPLACE ABVAR IN DVAR

    The following variable(s) will be changed:
    DVAR       10 5   ABCDABCYAB     ETX
    The following flag(s) will be set: None
```

Example:

```
    DESTIN     6  1   A1B2C3         ETX
               REPLACE "A1B2C3" IN DESTIN
```

```
    The following variable(s) will be changed:
    DESTIN    6  1   112233          ETX
    The following flag(s) will be set: None
```

Example:

```
    DESTIN    7  1   AEAFZAZ         ETX

              REPLACE "AZZA" IN DESTIN

    The following variable(s) will be changed:
    DESTIN    7  1   ZEZFAZA         ETX
    The following flag(s) will be set: None
```

Example:

```
    DESTIN    6  1   123456          ETX
    REPVAL    4  2   ABCD            ETX

              REPLACE REPVAL IN DESTIN

    The following variable(s) will be changed: None
    The following flag(s) will be set: EOS
```

CHAPTER 8. ARITHMETIC INSTRUCTIONS

The arithmetic instructions are used to perform the various arithmetic operations upon DATABUS operands. Generally all arithmetic instructions have the following form:

```
<label>  <oper>    <soper><prep><doper>
```

where: <label> is an execution label.
<oper> is the DATABUS arithmetic operation.
<soper> is the source operand.
<prep> is a valid preposition.
<doper> is the destination operand.

The DATABUS operation is performed using the source and destination operands. The result of the operation is generally transferred to the destination operand. The content of the source operand is never modified.

8.1 ADD

The ADD instruction causes the content of source operand to be added to the content of destination operand. The result (sum) is placed in the destination operand. This instruction has the following formats:

```
1)   <label>  ADD       <snvar><prep><dnvar>
2)   <label>  ADD       <nlit><prep><dnvar>
```

Where: <label> is an execution label (see section 2.).
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The source numeric operand is never modified.

-- <dnvar> contains the result (sum) of the ADD.

-- The flags OVER, LESS, ZERO (or EQUAL), are set appropriately.

-- The rounding and truncation rules are applicable (see section 2.7).

Examples:

```
        X       FORM    "123.45"
        Y       FORM    "267.22"

                ADD     X TO Y

                Y will contain 390.67
                The following flag(s) will be set: None
```

Example:

```
        CAT     FORM    "100.50"

                ADD     ".005" TO CAT

                CAT will contain 100.51
                The following flag(s) will be set: None
```

Example:

```
        NUM     FORM    "-245.0000"
        NUM2    FORM    "800.0"

                ADD     NUM TO NUM2

                NUM2 will contain 555.0
                The following flag(s) will be set: None
```

Example:

```
        N       FORM    "00.0"

                ADD     "100.00" TO N

                N will contain 00.0
                The following flag(s) will be set: OVER, ZERO
```

8.2 SUBTRACT (SUB)

The SUB instruction (The compiler will also accept a mnemonic of SUBTRACT) is used to perform a subtract operation. The contents of the source numeric operand (minuend) is subtracted from the destination numeric operand (subtrahend) and result (difference) is placed in the destination numeric operand.

The instruction has the following formats:

```
1)   <label>  SUB       <snvar><prep><dnvar>
2)   <label>  SUBTRACT  <snvar><prep><dnvar>
3)   <label>  SUB       <nlit><prep><dnvar>
4)   <label>  SUBTRACT  <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The flags OVER, LESS, ZERO (or EQUAL) are applicable.

-- The contents of the source operand is never modified.

-- The destination operand contains the result (difference).

-- The truncation and rounding rules apply.

Example:

```
A       FORM    "123.45"
B       FORM    "-20.45"

        SUB     B FROM A

        A will contain 143.90
        The following flags will be set: None
```

Example:

```
C1      FORM    "5.60"
C2      FORM    "1.665"

        SUB     C2 FROM C1

        C2 will contain 3.94
        The following flags will be set: None
```

Example:

```
NUMBR   FORM    "-345"

        SUB     "700.5" FROM NUMBR

        NUMBR will contain 1045
        The following flags will be set: OVER, LESS
```

Example:

```
Y1      FORM    " 10.00"
Y2      FORM    " 20.005"

        SUB     Y2 FROM Y1

        Y2 will contain -10.01
        The following flags will be set: LESS
```

8.3 MULTIPLY (MULT)

The MULT instruction (the compiler will also accept a mnemonic of MULTIPLY) causes the content of the source numeric (multiplicand) to be multiplied by the contents of the destination numeric operand (multiplier). The result (product) is placed in the destination numeric operand. The instruction has the following formats:

```
1)   <label>  MULT      <snvar><prep><dnvar>
2)   <label>  MULTIPLY  <snvar><prep><dnvar>
3)   <label>  MULT      <nlit><prep><dnvar>
4)   <label>  MULTIPLY  <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- The execution label <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The flags OVER, LESS, ZERO (or EQUAL) are applicable.

-- The source numeric operand is not modified.

-- The destination numeric operand contains the result (product).

-- The sum of the number of characters in the source operand and the destination operand must not exceed 31. (The compiler does not check this limit. If it is exceeded the interpreter will produce erroneous results.)

-- The truncation and rounding rules are applicable.

Example:

```
        M1      FORM    "010"
        M2      FORM    "012"

                MULT    M1 BY M2

                M2 will contain 120
                The following flag(s) will be set: None
```

Example:

```
        X123    FORM    "12000.00"

                MULT    "1.1" BY X123

                X123 will contain 13200.00
                The following flag(s) will be set: None
```

Example:

```
        NEG     FORM    "-10.5"

                MULT    "10" BY NEG

                NEG will contain 105.0

                The following flag(s) will be set: OVER, LESS
```

8.4 DIVIDE (DIV)

The DIV instruction (the compiler will also accept a mnemonic of DIVIDE) causes the content of the source operand (divisor) to be divided by the content of the destination operand (dividend). The result (quotient) is placed in the destination variable.

```
1)   <label>  DIV      <snvar><prep><dnvar>
2)   <label>  DIVIDE   <snvar><prep><dnvar>
3)   <label>  DIV      <nlit><prep><dnvar>
4)   <label>  DIVIDE   <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The contents of the source numeric operand (divisor) is not changed.

-- The contents of the destination numeric variable <dnvar> contains the result (quotient).

-- If the content of the source numeric operand is zero, then the OVER flag is set and the content of the destination numeric variable is determined by one of the following:

    1) If the source numeric operand (divisor) is an integer zero (contains no digits to the right of the decimal point) then the destination numeric variable (quotient) is set to the largest possible number that can be represented in the destination numeric variable.

    2) If the source numeric operand (divisor) is non-integer zero, then the destination numeric variable (quotient) is set to zero.

-- If the destination numeric variable (quotient) is not large

enough to contain the quotient, then one of the following actions is taken:

1) If the source numeric operand (divisor) is integer (no digits after the decimal point) the quotient will be truncated to the number of places in the destination numeric variable.

2) If the divisor is not integer (at least one decimal place after the decimal point) the quotient will be rounded to the number of places in the destination numeric variable.

-- There is a restriction on the length of division operands. The following formula is used to determine acceptable lengths (Decimal points are not counted as characters when using the following formula).

N=2*NR+NU+NL

Where: NR is the number of digits after the decimal point in the divisor.

NU is the number of characters in the dividend.

NL is the number of characters in the divisor.

"*" represents multiplication.

N computed by the above formula must be less than 32. The compiler does not check this limit. If it is exceeded the interpreter will produce erroneous results.

-- The flags OVER, LESS, ZERO (or EQUAL) are applicable.

-- The truncation and rounding rules apply.

Example:

```
ONEH    FORM    "100.00"
TEN     FORM    "10"

        DIV     TEN INTO ONEH

        ONEH contains 10.00
        The following flag(s) are set: None
```

Example:

```
        ZERO    FORM    "000"
        N       FORM    "155.00"

                DIV     ZERO INTO N

                N will contain 999.99
                The following flag(s) will be set: OVER
```

Example:

```
        ZERO    FORM    "00.00"
        N       FORM    "155.00"

                DIV     ZERO INTO N

                N will contain __.00
                The following flag(s) will be set: OVER, ZERO
```

Example:

```
        N1      FORM    "100"

                DIV     "0.1" INTO N1

                N1 will contain __0
                The following flag(s) will be set: OVER, ZERO
```

8.5 MOVE

The MOVE instruction causes the content of the source numeric operand to replace the content of the destination numeric operand. The instruction has the following formats:

```
1)   <label>  MOVE      <snvar><prep><dnvar>
2)   <label>  MOVE      <nlit><prep><dnvar>
```

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- The contents of the source numeric operand is never modified.

-- The destination numeric variable contains the result of the MOVE operation.

-- The OVER, LESS, ZERO (or EQUAL) flags are applicable.

-- The truncation and rounding rules are applicable.

Example:

```
        SOURCE  FORM    "12345"
        DESTIN  FORM    6.2

                MOVE    SOURCE TO DESTIN

                DESTIN will contain _12345.00
                The following flag(s) will be set: None
```

Example:

```
        D1      FORM    4.2
                MOVE "12345" TO D1

                D1 will contain 2345.00
                The following flag(s) will be set: OVER
```

Example:

```
        S       FORM    "12345.51"
        D       FORM    "99999"

                MOVE    S TO D

                D will contain 12346
                The following flag(s) will be set: None
```

Example:

```
        N       FORM    "999.99"

                MOVE    "0.0" TO N
```

N will contain ___.00
The following flag(s) will be set: ZERO

8.6 COMPARE

The COMPARE instruction is used to compare two numeric quantities. The instruction has the following formats:

1) <label> COMPARE <snvar><prep><dnvar>
2) <label> COMPARE <nlit><prep><dnvar>

Where: <label> is an execution label.
<snvar> is the source numeric variable.
<prep> is a preposition.
<dnvar> is the destination numeric variable.
<nlit> is a numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> is a valid numeric literal.

-- The contents of the source numeric operand are never modified.

-- The contents of the destination numeric variable are never modified.

-- The LESS and ZERO (or EQUAL) condition flags are set exactly as if a SUBTRACT instruction had been executed instead of a COMPARE.

-- Rounding takes place when the COMPARE instruction is executed.

-- Since the result is not moved to the destination variable, the format of the result is not taken into consideration when setting the condition flags. This means that the OVER condition flag can never be set by the COMPARE instruction.

Example:

```
ONEH    FORM    "100.00"

        COMPARE "100" TO ONEH

        The following flag(s) will be set: ZERO (EQUAL)
```

Example:

```
        OP1     FORM    "0100.0"
        OP2     FORM    "090"

                COMPARE OP1 TO OP2

                The following flag(s) will be set: LESS
```

Example:

```
        CAT     FORM    "999"

                COMPARE "-1" TO CAT

                The following flag(s) will be set: none
```

Example:

```
        F       FORM    "-99"

                COMPARE "1" TO F

                The following flag(s) will be set: LESS
```

8.7 LOAD

The LOAD instruction selects (using an index for selection) a numeric variable from a list using an index and performs a MOVE operation on the selected numeric variable to the destination numeric variable. The instruction has the following formats:

```
  <label>  LOAD      <dnvar><prep><index><prep><list>
```

Where: <label> is execution label.
<dnvar> is the destination numeric variable.
<prep> is a preposition.
<index> is a numeric variable which specifies which item of the available list is to be selected.

<list> is a list of numeric variables.

Programming Considerations:

-- <label> is optional.

-- <dnvar> contains the result of the LOAD instruction after execution.

-- <index> is a numeric variable which specifies which item from the available list should be selected. If the index is not an integer, the index is truncated, and the integer portion is used for list selection. An index numeric variable of one (1) specifies the first item in the list and an index value of n specifies the nth item in the list.

-- If the index contains a number which does not correspond to one of the list items, then the LOAD instruction is ignored and execution continues with the next DATABUS instruction.

-- <list> may be continued on the following line by use of the colon (:).

-- The <index> is not modified.

-- None of the <list> items are modified.

-- The OVER, LESS, ZERO (or EQUAL) flags are applicable.

-- The truncation and rounding rules are used.

Example:

```
DESTIN  FORM    "9999"
INDEX   FORM    "2"
X1      FORM    "1111"
X2      FORM    "2222"
X3      FORM    "3333"

        LOAD    DESTIN FROM INDEX OF X1,X2,X3

        DESTIN will contain 2222
        The following flag(s) will be set: None
```

Example:

```
Y       FORM    3.1
I       FORM    "1.6"
S1      FORM    "-11.36"
S2      FORM    "222"
S3      FORM    "333"

        LOAD    Y FROM I OF S1,S2:
```

```
                    S3

          Y will contain -11.4
          The following flag(s) will be set: LESS
```

8.8 STORE

The STORE instruction selects (using an index for selection) a numeric variable from a list and performs a MOVE operation from the source numeric operand to the selected destination numeric variable. The instruction has the following formats:

```
1)   <label>  STORE     <snvar><prep><index><prep><list>
2)   <label>  STORE     <nlit><prep><index><prep><list>
```

Where:
- <label> is an execution label.
- <snvar> is the source numeric variable.
- <index> is the index numeric variable which specifies which item from the available list is to be selected.
- <prep> is a preposition.
- <list> is a list of numeric variables.
- <nlit> is numeric literal.

Programming Considerations:

-- <label> is optional.

-- <nlit> must be a valid numeric literal.

-- <dnvar> contains the result of the STORE operation.

-- <index> is a numeric variable which specifies which item from the available list should be selected. If the index is not an integer, the index is truncated, and the integer portion is used for list selection. An index numeric variable of one (1) specifies the first item in the list and an index value of n specifies the nth item in the list.

-- If the index contains a number which does not correspond to one of the list items, then the STORE instruction is ignored and execution continues with the next DATABUS instruction.

-- The variables contained in <list> are separated by a comma (,).

-- <list> may be continued on the following line by use of the colon (:) in place of the comma after the last variable on the line to be continued.

-- The <index> is never modified.

-- Only the selected numeric variable from the <list> is modified.

-- The OVER, LESS, ZERO (or EQUAL) flags are applicable.

-- The truncation and rounding rules apply.

Example:

```
SOURCE  FORM    "999"
INDEX   FORM    "1.9"
D1      FORM    "111"
D2      FORM    "222"
D3      FORM    "333"

        STORE   SOURCE INTO INDEX OF D1,D2:
                D3
```

D1 will contain 999. The other variables D2 and D3 will be unchanged.
The following flag(s) will be set: None

Example:

```
SOURCE  FORM    "1234"
I       FORM    "4"
D1      FORM    4
D2      FORM    4

        STORE   SOURCE INTO I OF D1,D2
```

The contents of neither D1 or D2 is changed because the index was out of range.
The following flag(s) are set: None

8.9 CHECK11 (CK11)

The CHECK11 (the compiler will also accept a mnemonic of CK11) instruction performs a modulo 11 check digit calculation on two string variables. The instruction has the following formats:

```
1)   <label>  CHECK11   <svar1><prep><svar2>
2)   <label>  CK11      <svar1><prep><svar2>
3)   <label>  CHECK11   <svar1><prep><slit>
4)   <label>  CK11      <svar1><prep><slit>
```

Where: <label> is an execution label.
<svar1> is a string variable called the base string which contains the base number and the check digit.
<prep> is a preposition.
<svar2> is a string variable which contains the weighting factor.
<slit> is a string literal.

The following algorithm is used to perform the CHECK11 instruction.

Let the length n of the base string be defined as n=LL-FP where:

LL=logical length of base string.

FP=formpointer of base string.

The base string is composed of two parts:

1) The base number which is the first n digits of the base string.

2) The check digit which is the digit following the base number.

Let the individual digits of the base number be b(1), b(2),...,b(n) where b(1) is the formpointed left most digit, and b(n) is the right most digit of the base number.

Let the individual digits of the weighting factor be w(1), w(2)...,w(n) with w(1) the formpointed left most digit and w(n) is the right most digit of the weighting factor.

The following sum S is formed.

S=b(1)*w(1)+b(2)*w(2)+...+b(n)*w(n)

Then the computed check digit C is:

C=11-R(S/11) where R(S/11) is the remainder from the division S/11.

The computed check digit C is compared to the check digit supplied in the base string. If they are equal, the EQUAL flag is set, otherwise the OVER flag is set and the EQUAL flag cleared.

Programming Considerations:

-- <label> is optional.

-- Neither of the variables <svar1> or <svar2> is modified.

-- <svar1>, <svar2>, and <slit> when used must contain digits only.

-- If the length (LL-FP) of the weighting factor is less than the length n of the base number, then the OVER flag is set and the DATABUS instruction is not finished.

-- A computed check digit with a value of 10 or greater cannot be used and causes the OVER flag to be set.

Example:

```
BASSTR  INIT    "12343"
WEIGHT  INIT    "5432"

        CHECK11 BASSTR BY WEIGHT

        The following flag(s) are set: ZERO (EQUAL)
```

Example:

```
BASSTR  INIT    "12342"
WEIGHT  INIT    "654"


        RESET   BASSTR TO 3
        RESET   WEIGHT TO 2
        CHECK11 BASSTR BY WEIGHT

        The following flag(s) are set: ZERO (EQUAL)
```

Example:

```
B       INIT    "141599"
W       INIT    "41"


        RESET   B TO 4
        LENSET  B
        RESET   B TO 2
        CHECK11 B BY W

        The following flag(s) are set: ZERO (EQUAL)
```

Example:

```
B       INIT    "141699"
W       INIT    "41"

        RESET   B TO 4
        LENSET  B
        RESET   B TO 2
        CHECK11 B BY W

        The following flag(s) are set: OVER
```

8.10 CHECK10 (CK10)

The CHECK10 (the compiler will also accept a mnemonic of CK10) instruction performs a modulo 10 check digit calculation on two string variables. The instruction has the following formats:

```
1)   <label>  CHECK10   <svar1><prep><svar2>
2)   <label>  CK10      <svar1><prep><svar2>
3)   <label>  CHECK10   <svar1><prep><slit>
4)   <label>  CK10      <svar1><prep><slit>
```

Where: <label> is an execution label.
<svar1> is a string variable called the base string which contains the base number and the check digit.
<prep> is a preposition.
<svar2> is a string variable which contains the weighting factor.
<slit> is a string literal which contains the weighting factor.

The following algorithm is used to perform the CHECK10

instruction.

Let the length of the base string be defined as n=LL-FP where:

LL=Logical length of base string.

FP=formpointer of base string.

The base string is composed of two parts:

1) The base number which is the first n digits of the base string.

2) The check digit which is the digit following the base number.

Let the individual digits of the base number be b(1), b(2),...b(n) where b(1) is the formpointed left most digit, and b(n) is the right most digit of the base number.

Let the individual digits of the weighting factor be w(1), w(2)...,w(n) with w(1) the formpointed left most digit and w(n) is the right most digit of the weighting factor.

Let the following products be formed:

```
P(1) = b(1)*w(1)
P(2) = b(2)*w(2)
.
. etc.
.
P(n) = b(n)*w(n)
```

Take each P(i) and perform a "lateral" addition on the individual digits (i.e. P(3)=32 would yield a "lateral addition" of 5). Let the "lateral" addition of the digits of each P(i) be S(i). Then form the following sum:

SD=S(1)+S(2)+...+S(i)

Then the computed check digit C is:

C=10-R(SD/10) Where R(SD/10) is the remainder from the division SD/10.

The computed check digit C is compared to the check digit

supplied in the base string. If they are equal, the EQUAL flag is set, otherwise the OVER flag is set and the EQUAL flag is cleared.

Programming Considerations:

-- <label> is optional.

-- Neither of the variables <svar1> or <svar2> is modified.

-- <svar1>, <svar2>, and <slit> when used must contain digits.

-- If the length (LL-FP) of the weighting factor is less than the length n of the base number, then the OVER flag is set and the DATABUS instruction is not finished.

-- If a computed check digit of 10 is used, it is treated modulo 10.

Example:

```
X       INIT    "12340"
Y       INIT    "5432"

        CHECK10 X BY Y

        The following flag(s) are set: EQUAL
```

Example:

```
BASE    INIT    "1515999"


        RESET   BASE TO 4
        LENSET  BASE
        RESET   BASE
        CHECK10 BASE BY "515"

        The following flag(s) are set: EQUAL
```

Example:

```
BASE    INIT    "9653"
WEIGHT  INIT    "521"

        CHECK10 BASE BY WEIGHT
```

The following flag(s) are set: EQUAL

Example:

```
BASE    INIT    "1650"
WEIGHT  INIT    "121"

        CHECK10 BASE BY WEIGHT

The following flag(s) are set: OVER
```

CHAPTER 9. INTERACTIVE INPUT/OUTPUT

These instructions are used to input from a keyboard and output to a CRT screen (or output to any device used in place of the CRT screen).

General Programming Considerations:

-- Typically, formatting is handled in one of the following ways.

   a) By the way a variable is defined. It should be defined with the format which is to be used for input/output.

   b) Using list controls.

-- Normally, when execution of one of these I/O statements terminates, the cursor position is reset to the beginning of the next line.

-- If a semicolon is used after the last item in the list, the cursor position remains where it was on statement termination. This feature allows a second I/O statement to continue where the first statement left off.

   Example:

```
            DISPLAY   "FLAGS: ";
            CALL      NOTFLG IF NOT ZERO
            DISPLAY   "ZERO, ";
            CALL      NOTFLG IF NOT LESS
            DISPLAY   "LESS"
            ...       ...
  NOTFLG    DISPLAY   "NOT ";
            RETURN
```

   would display <u>one</u> of the following lines, depending on the condition flags.

```
FLAGS: ZERO, LESS
FLAGS: ZERO, NOT LESS
FLAGS: NOT ZERO, LESS
FLAGS: NOT ZERO, NOT LESS
```

-- Those instructions that use a list should make use of continuation when it is possible to do so. (For details about

using continuation, see section 2.) This not only increases the execution speed of the program, but also decreases the system overhead. The programmer should check his program for any occurance of two consecutive I/O instructions that are the same. These two instructions can be replaced with a single instruction by using continuation.

Example:

```
DISPLAY   "LINE ONE"
DISPLAY   "LINE TWO"
```

should be combined to form the statement below.

```
DISPLAY   "LINE ONE":
          *N,"LINE TWO"
```

-- The condition flags are unchanged by the execution of these statements.

9.1 KEYIN

KEYIN is used primarily to input from the keyboard, though in some cases it can be used to output to the screen. This statement has the following general format:

```
<label>  KEYIN     <list>
```

where: <label> is an execution label (see section 2.).
<list> is a list of items describing the input from the keyboard.

Programming Considerations:

-- <label> is optional.

-- The items in the list must be separated by commas.

-- <list> may be made up of any combination of the following items:

a) <svar>, a character string variable (see section 4.2).

b) <nvar>, a numeric string variable (see section 4.1).

c) <occ>, an octal control character (see section 2.5).

d) <list control>, used to control the manner in which the list is processed.

e) <slit>, a literal of the form "<string>" (see section 2.5). <string> must be a valid character string (see section 4.2).

f) <nlit>, a literal of the form "<string>" (see section 2.5). <string> must be a valid numeric string (see section 4.1).

9.1.1 Character String Variables (KEYIN)

When a character string variable (<svar>) appears in the list of a KEYIN instruction, characters are accepted from the keyboard and put into the variable. Unless modified by a list control, the manner in which the characters are accepted is described below.

Programming Considerations:

-- When characters are being accepted from the keyboard, the flashing cursor is on. At all other times the cursor is off. (The *EOFF list control, see section 9.1.3.13, will cancel this.)

-- Only ASCII characters are accepted.

-- Each character, as it is accepted, is displayed on the screen.

-- The horizontal cursor position is bumped by 1 for each character accepted.

-- Characters are stored consecutively starting at the physical beginning of the string.

-- Characters are accepted up to the physical length of the character string variable.

-- A beep is sounded at the terminal for each character that will not fit within the variable.

-- If a null string is entered (if the ENTER key is struck without any other characters having been entered),

a) the formpointer of the variable is set to zero.

b) the logical length pointer of the variable is set to zero.

c) the value of the variable will be indeterminate.

To check for a null string entry; the program can first execute a RESET or CMATCH using the variable in question, then check the EOS condition flag.

-- If the string entered is not null,

a) the formpointer of the variable is set to one.

b) the logical length pointer of the variable is set to the last character entered.

c) the suffix of the string variable is unchanged.

-- Processing is continued with the next item in the list when the ENTER key is struck. (See section 9.1.5.2 on the NEWLINE key.)

9.1.2 Numeric String Variables (KEYIN)

When a numeric string variable (<nvar>) appears in the list of a KEYIN instruction, characters are accepted from the keyboard and put into the variable. Unless modified by a list control, the manner in which the characters are accepted is described below.

Programming Considerations:

-- When characters are being accepted from the keyboard, the flashing cursor is on. At all other times the cursor is off.

-- Each character, as it is accepted, is displayed on the screen.

-- The horizontal cursor position is bumped by 1 for each character accepted.

-- The following depend on the format of the numeric variable:

a) A minus sign is accepted only if it is the first character entered.

b) A minus sign is accepted only if there is room for at least one character to the left of the decimal point.

c) A period is accepted only if the format calls for a decimal point.

d) Only one period will be accepted.

e) The number of characters that will be accepted before a period is required, is equal to the number of places preceding the decimal point in the format of the variable.

f) The number of characters that will be accepted after the period is equal to the number of places following the decimal point in the format of the variable.

g) If the ENTER key is the first key struck, a value of zero is entered.

-- If a character is entered that is not acceptable to the format of the numeric variable, a beep is sounded at the terminal.

-- The number entered will be reformatted to match the format of the variable when the ENTER key is struck (see section 4.1).

-- Processing is continued with the next item in the list when the ENTER key is struck.

Example: If the following statement is used to define NVAR;

```
NVAR     FORM      2.1
```

then when NVAR is used in a KEYIN statement, the following characters will result in NVAR having the values shown.

ascii	ascii	ascii	ascii	ascii	value of NVAR
ENTER					.0
.	ENTER				.0
.	2	ENTER			.2
-	.	ENTER			-.0
-	.	2	ENTER		-.2
-	2	ENTER			-2.0
-	2	.	ENTER		-2.0
-	2	.	3	ENTER	-2.3
2	ENTER				2.0
2	.	ENTER			2.0
2	.	3	ENTER		2.3
2	3	ENTER			23.0
2	3	.	ENTER		23.0
2	3	.	4	ENTER	23.4

9.1.3 List Controls

The list controls are provided to allow more flexibility for data entry. They may be used to control the manner in which data is requested and input into variables. All list controls begin with an asterisk followed by the specification of the control function.

9.1.3.1 *P<h>:<v> (Cursor Positioning)

This list control is used to position the cursor on the screen. The following is the general format of this control.

*P<h>:<v>

where: <h> is the horizontal cursor position.
<v> is the vertical cursor position.

Programming Considerations:

-- <h> and <v> may be any combination of the following:

a. <dnum>, where <dnum> is a decimal number.

b. <nvar>, where <nvar> is a numeric variable (see section 4.1).

c. <evar>, where <evar> is a data label defined using an EQUATE (see section 3.1).

-- Both <h> and <v> must be specified.

-- The value of <h> should be between 1 and 80. (See the user's guide of the appropriate interpreter for any exceptions or differences.) Positions outside this range are reset to the largest value of the range.

-- The value of <v> should be between 1 and 24. (See the user's guide of the appropriate interpreter for any exceptions or differences.) Positions outside this range are reset to the largest value of the range.

9.1.3.2 *EL (Erase to the End-of-Line)

The *EL control causes the line to be erased starting with the current cursor position and continuing to the right. The cursor position is unchanged by the execution of this control.

Example:

```
KEYIN     *P50:10,*EL,"OK? (Y/N) ",REPLY
```

This statement would erase line 10, starting with column 50.

9.1.3.3 *EF (Erase from Cursor Position)

The *EF control performs the function of *EL and additionally erases all screen lines below the current cursor position. The cursor position is unchanged by the execution of this control.

Example:

```
KEYIN     *P50:20,*EF
```

This statement would produce the same results as the following statement.

```
KEYIN     *P50:20,*EL:
          *P1:21,*EL:
          *P1:22,*EL:
          *P1:23,*EL:
          *P1:24,*EL:
          *P50:20
```

9.1.3.4 *ES (Erase the Screen)

The *ES control positions the cursor to 1:1 and erases the entire screen. The cursor is left positioned to 1:1.

Example:

```
KEYIN     *ES
```

Executing the above statement is equivalent to executing the following statement.

```
KEYIN     *P1:1,*EF
```

9.1.3.5 *C (Carriage Return)

The *C control causes the cursor to be set to the beginning of the current line. For example: if the cursor was positioned to 40:5, executing the *C control would change the cursor position to 1:5.

9.1.3.6 *L (Line Feed)

The *L control causes the cursor to be set to the following line in the current horizontal position. For example: if the cursor was positioned to 20:5, executing the *L control would change the cursor position to 20:6.

9.1.3.7 *N (Next Line)

The *N control causes the cursor to be set to the first column of the next line. Executing the *N control is equivalent to executing a *C control followed by a *L control.

9.1.3.8 *R (Roll the Screen)

The *R control causes the screen to roll up by one line. (This control has no effect when sent to a 3360 terminal. It is included for use with 3600 terminals and the system console.) The cursor position is unchanged by the execution of this control.

9.1.3.9 *+ (KEYIN Continuous On)

This control is used to turn on a mode of entry called keyin continuous. This mode allows the system to react in much the same way as a keypunch machine that is using a control card.

Programming Considerations:

-- This control affects data entry of all variables which follow the *+ control in the KEYIN list.

-- If keyin continuous is turned on; entering the last character acceptable to the format of a variable will cause the system to react as if the ENTER key had been struck.

-- Keyin continuous may be turned off by the use of the *- list control (see section 9.1.3.10).

-- Keyin continuous is automatically turned off when the end of the KEYIN list is reached.

9.1.3.10 *- (KEYIN Continuous Off)

This control turns the keyin continuous mode off. For more details about the keyin continuous mode, see section 9.1.3.9.

9.1.3.11 *T (KEYIN Timeout)

This control causes a time out if the time between entering two characters is too long.

Programming Considerations:

-- This control causes a time out if more than two seconds elapse between entering any two characters.

-- If a time out occurs, the remainder of the KEYIN list is treated as though the NEW LINE key had been struck. (For more details about NEW LINE, see section 9.1.5.2.)

9.1.3.12 *W (Wait)

This control is an effective way of allowing a program to pause without imposing significant overhead on the system.

Programming Considerations:

-- Each occurance of a *W in the KEYIN list causes a pause of one second before continuing to the next item in the list.

-- Any number of seconds of pause may be achieved by simply putting in the required number of *W controls in the list.

9.1.3.13 *EOFF (Echo Off)

This control is used to suppress the character display (echo) of all characters accepted from the keyboard. This is useful in message switching applications or for entry of passwords or other security information.

Programming Considerations:

-- This control causes echo suppression for all variables which follow the *EOFF in the KEYIN list.

-- The beep returned when an invalid character is entered is also suppressed by this control.

-- The echo may be re-enabled by using the *EON list control (see section 9.1.3.14).

-- The echo is re-enabled when the end of the KEYIN list is reached.

Example: The following KEYIN statement could be used to enter a password.

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

9.1.3.14 *EON (Echo On)

This control is used to re-enable the echoing of characters to the screen while entering data. For more details on echo suppression see section 9.1.3.13.

9.1.3.15 *IT (Invert Text)

This control is used to disable shift key inversion. The normal state of the keyboard is with shift key inversion enabled. This means that all lower case alphabetic characters are entered and displayed as upper case characters and vice versa. Shift key inversion disabled is the normal state of a typewriter; that is, the shift key must be used to get upper case alphabetic characters.

Programming Considerations:

-- Shift key inversion is only useful on those terminals that have both an upper and lower case character set. For instance, the Datapoint 3360 cannot make use of shift key inversion while the Datapoint 3600 can.

-- Shift key inversion affects only the alphabetic characters and not the numerals or punctuation.

-- The *IT control causes any letter entered with the SHIFT key depressed to be entered and displayed as an upper case letter.

-- Shift key inversion will remain disabled until a *IN control is used (see section 9.1.3.16).

-- Shift key inversion will be enabled when a CHAIN instruction is executed (see section 6.8).

9.1.3.16 *IN (Invert to Normal)

This control is used to enable shift key inversion. For more details on shift key inversion, see section 9.1.3.15.

Programming Considerations:

-- Shift key inversion is only useful on those terminals that have both an upper and lower case character set. For instance, the Datapoint 3360 cannot make use of shift key inversion while the Datapoint 3600 can.

-- Shift key inversion affects only the alphabetic characters and not the numerals or punctuation.

-- The *IN control causes any letter entered with the SHIFT key depressed to be entered and displayed as a lower case letter.

-- Shift key inversion will remain enabled until a *IT control is used (see section 9.1.3.15).

-- Shift key inversion will be enabled when a CHAIN instruction is executed (see section 6.8).

9.1.3.17 *JL (Justify Left)

This control is used to cause the characters entered into a variable to be left justified within that variable.

Programming Considerations:

-- This control affects only the first variable following the *JL in the KEYIN list.

-- When the variable affected by the *JL is a numeric string variable, the following are true.

    a) If a decimal point is not entered,

        1) all digits entered are put into the leftmost positions

of the numeric variable.

2) all remaining character positions of the variable are filled with zeros.

b) If a decimal point is entered, the *JL control has no effect on the numeric variable.

-- When the variable affected by the *JL is a character string variable, the following are true.

a) The variable is first filled with blanks.

b) The characters entered from the keyboard are put into the variable normally (see section 9.1.1).

c) The logical length pointer points to the last physical character in the variable.

-- This control may be used in conjunction with the *DE control (see section 9.1.3.20).

Example: If the following statements are used to define SVAR and NVAR;

```
NVAR     FORM     3.3
SVAR     DIM      5
```

then when NVAR and SVAR are used in a KEYIN statement with *JL, the following characters will result in the variables having the values shown below. The underline character (_) is used to indicate a blank.

ascii	ascii	ascii	ascii	ascii	value of NVAR	value of SVAR
1	2	ENTER			120.000	12___
1	2	.	ENTER		12.000	12.__
1	ENTER				100.000	1____
-	1	ENTER			-10.000	-1___
-	1	.	ENTER		-1.000	-1.__

9.1.3.18 *JR (Justify Right)

This control is used to cause the characters entered into a character string variable to be right justified within that variable.

Programming Considerations:

-- This control affects only the first variable following the *JR in the KEYIN list.

-- If a null string is entered (ENTER is the first character entered):

   a) The variable is filled with blanks.

   b) The formpointer is set to zero.

   c) The logical length pointer is set to zero.

-- If the string entered is not null:

   a) The characters entered are right justified within the variable. This means that, when the characters are put into the variable, they are all shifted to the right until the rightmost character entered is put into the rightmost character position in the variable.

   b) All character positions that are vacated when the string is right justified are filled with blanks.

   c) The formpointer points to the first physical character of the variable.

   d) The logical length pointer points to the last physical character of the variable.

-- This control may be used in conjunction with:

   a) the *ZF control (see section 9.1.3.19). When *ZF and *JR are used together:

      1) Any characters entered are right justified with zero fill.

      2) A null entry will first fill the variable with zeros, then set the formpointer and logical length pointer to zero.

b) the *DE control (see section 9.1.3.20).

Example: If the following statement is used to define SVAR;

```
SVAR     DIM      5
```

then when SVAR is used in a KEYIN statement with *JR, the following characters will result in SVAR having the values shown below. The underline character (_) is used to indicate a blank.

ascii	ascii	ascii	ascii	ascii	ascii	value of SVAR
1	2	3	ENTER			__123
1	2	3	4	ENTER		_1234
1	2	3	4	.	ENTER	1234.
1	2	3	.	ENTER		_123.
1	2	.	3	ENTER		_12.3
1	.	2	3	ENTER		_1.23
A	B	C	ENTER			__ABC

9.1.3.19 *ZF (Zero Fill)

This control is used to cause a character string variable to be zero filled.

Programming Considerations:

-- This control is the same as the *JL control (see section 9.1.3.17) with the following exceptions:

a) *ZF applies only to character string variables.

b) The variable is filled with zeros instead of blanks.

-- This control may be used in conjunction with:

a) the *JR control (see section 9.1.3.18). When *ZF and *JR are used together:

1) Any characters entered are right justified with zero fill.

2) A null entry will first fill the variable with zeros, then set the formpointer and logical length pointer to zero.

b) the *DE control (see section 9.1.3.20).

9.1.3.20 *DE (Digit Entry)

This control may be used to restrict input into a character string variable to digits only (0-9).

Programming Considerations:

-- This control affects only the first variable following the *DE in the KEYIN list.

-- An attempt to enter a non-digit will result in the character being ignored and a beep being returned.

-- This control may be used in conjunction with:

   a) the *JL control (see section 9.1.3.17).

   b) the *JR control (see section 9.1.3.18).

   c) the *ZF control (see section 9.1.3.19).

9.1.4 Literals (KEYIN)

When a literal (<occ>, <slit> or <nlit>) appears in the list of a KEYIN statement, that literal is displayed on the screen.

Programming Considerations:

-- If the literal is an octal control character (see section 2.5), it is sent to the terminal.

-- If the literal is of the form "<string>", the following rules apply.

   a) All of the characters between the double quotes are displayed as they appear in the literal.

   b) The first character of the string is displayed at the current cursor position.

   c) The cursor is bumped one position to the right for every character displayed.

   d) The cursor is left positioned one position to the right of the last character of the literal.

9.1.5 Special Considerations

The following sections describe some special cases of operator input from the keyboard.

9.1.5.1 BACKSPACE and CANCEL

The following special keys are useful in correcting typing errors while entering data into variables that appear in a KEYIN list.

-- The BACKSPACE key (control H on Teletype) may be used to delete the last character entered. Using BACKSPACE causes the following actions:

   a) The cursor is moved one position to the left.

   b) The character under the cursor is erased from the screen.

   c) The character that was under the cursor is not deleted from the variable in the KEYIN list. The KEYIN pointers are decremented by one without restoring the original contents of the variable.

-- The CANCEL key (control X on a Teletype) may be used to reset KEYIN pointers to the beginning of the variable.

   Using CANCEL is like performing repeated BACKSPACE's until the variable has been cleared.

-- Neither BACKSPACE nor CANCEL overstore the contents of the variable with blanks.

-- Once BACKSPACE or CANCEL has been used the contents of the variable becomes indeterminate.

9.1.5.2 NEW LINE

Using the NEW LINE character is treated as a special case of using the ENTER character. Using the NEW LINE character effectively causes an automatic ENTER for all subsequent variables in the KEYIN list. The NEW LINE character is entered by striking:

a) the NEW LINE key on Datapoint 3360 and 3600 terminals,

b) control O on a Teletype, or

c) the DEL key (shift underline) on the system console.

Programming Considerations:

-- Using NEW LINE causes data entry into the current variable to be terminated as if the ENTER key had been struck instead.

-- All subsequent character string variables in the KEYIN list have their formpointer and logical length pointer set to zero.

-- All subsequent numeric string variables in the KEYIN list are set to zero.

-- The KEYIN list is processed normally, except for the variables, which are handled as stated above.

-- Control will fall through to the next Databus statement.

9.1.5.3 INTerrupt

Entering the INTerrupt character may be used to cause an immediate CHAIN to the port's MASTER program (see section 6.8). This allows a program to be interrupted before it runs to completion. The INTerrupt character is entered by striking:

a) the INT key on Datapoint 3360 and 3600 terminals,

b) control shift L on a Teletype, or

c) the CANCEL key with both the KEYBOARD and DISPLAY keys depressed on the system console.

Programming Considerations:

-- The program that is being interrupted will execute the equivalent of a STOP instruction (see section 6.7).

-- If the PI instruction (see section 6.12) is in effect at the time that an INTerrupt occurs, the interrupt procedure will be postponed.

-- If the printer is being used by the port receiving the INTerrupt, it will be RELEASEd (see section 10.3).

9.2 DISPLAY

The DISPLAY instruction is used to put information on the terminal screen. This statement has the following general format:

<label> DISPLAY <list>

where: <label> is an execution label (see section 2.).
<list> is a list of items describing the information to be put on the screen.

Programming Considerations:

-- <label> is optional.

-- The items in the list must be separated by commas.

-- <list> may be made up of any combination of the following items:

a) <svar> is a character string variable (see section 4.2).

b) <nvar> is a numeric string variable (see section 4.1).

c) <occ> is an octal control character (see section 2.5).

d) <list control> is used to control the manner in which the list is processed.

e) <slit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid character string (see section 4.2).

f) <nlit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid numeric string (see section 4.1).

9.2.1 Character String Variables (DISPLAY)

When a character string variable (<svar>) appears in the list of a DISPLAY instruction, the characters saved in the variable are displayed on the screen. Unless modified by a list control the manner in which the characters are put on the screen is described below.

Programming Considerations:

-- The characters in the variable are displayed starting with the first physical character and continuing through the logical length.

-- Blanks will be displayed for any character positions that exist between the logical length pointer and the physical end of the variable.

-- The first character displayed will be displayed at the current cursor position.

-- The horizontal cursor position is bumped by 1 for each character displayed.

-- The cursor is left positioned one character to the right of the last character displayed.

9.2.2 Numeric String Variables (DISPLAY)

When a numeric string variable (<nvar>) appears in the list of a DISPLAY instruction, the characters that are saved in the variable are displayed on the screen. Unless modified by a list control, the manner in which the characters are displayed is described below.

Programming Considerations:

-- The characters displayed start with the first physical character and continue through the physical end of the variable.

-- The first character displayed will be displayed at the current cursor position.

-- The horizontal cursor position is bumped by 1 for each character displayed.

-- The cursor is left positioned one character to the right of the last character displayed.

9.2.3 List Controls

The list controls are provided to allow more flexibility in the way the screen is formatted. They may be used to control the manner in which variables are displayed on the screen. All list controls begin with an asterisk followed by the specification of the control function.

9.2.3.1 *P<h>:<v> (Cursor Positioning)

This list control is used to position the cursor on the screen. For details on using this control, see section 9.1.3.1.

9.2.3.2 *EL (Erase to End-of-Line)

The *EL control causes the line to be erased to the right of the cursor position. For details on using this control, see section 9.1.3.2.

9.2.3.3 *EF (Erase to End-of-Frame)

The *EF control erases the screen from the cursor position to the bottom of the screen. For details on using this control, see section 9.1.3.3.

9.2.3.4 *ES (Erase the Screen)

The *ES control positions the cursor to 1:1 and erases the entire screen. For details on using this control, see section 9.1.3.4.

9.2.3.5 *C (Carriage Return)

The *C control causes the cursor to be set to the beginning of the current line. For example: if the cursor was positioned to 40:5, executing the *C control would change the cursor position to 1:5.

9.2.3.6 *L (Line Feed)

The *L control causes the cursor to be set to the following line in the current horizontal position. For example: if the cursor was positioned to 20:5, executing the *L control would change the cursor position to 20:6.

9.2.3.7 *N (Next Line)

The *N control causes the cursor to be set to the first column of the next line. Executing the *N control is equivalent to executing a *C control followed by a *L control.

9.2.3.8 *R (Roll the Screen)

The *R control causes the screen to roll up by one line. (This control has no effect when sent to a 3360 terminal. It is included for use with 3600 terminals and the system console.) The cursor position is unchanged by the execution of this control.

9.2.3.9 *+ (DISPLAY Blank Suppression On)

This control is used to turn on a display mode called blank suppression.

Programming Considerations:

-- This control affects the display of all character string variables which follow the *+ control in the DISPLAY list.

-- If blank suppression is turned on, character string variables are displayed on the screen as described below.

   a) The characters in the variable are displayed starting with the first physical character and continuing through the logical length.

   b) The first character will be displayed at the current cursor position.

   c) The horizontal cursor position is bumped by 1 for each character displayed.

   d) The cursor is left positioned one character to the right of the last character displayed.

-- Blank suppression is automatically turned off when the end of the DISPLAY list is reached.

9.2.3.10 *- (DISPLAY Blank Suppression Off)

This control turns blank suppression mode off. For more details about blank suppression mode, see section 9.2.3.9.

9.2.3.11 *W (Wait)

This control is an effective way of allowing a program to pause without imposing significant overhead on the system.

Programming Considerations:

-- Each occurance of a *W in the DISPLAY list causes a pause of one second before continuing to the next item in the list.

-- Any number of seconds of pause may be achieved by simply putting in the required number of *W controls in the list.

9.2.3.12 *IT (Invert Text)

This control is used to disable shift key inversion. For details on using this control, see section 9.1.3.15.

9.2.3.13 *IN (Invert to Normal)

This control is used to enable shift key inversion. For details on using this control, see section 9.1.3.16.

9.2.4 Literals (DISPLAY)

When a literal (<occ>, <slit> or <nlit>) appears in the list of a DISPLAY statement, that literal is displayed on the screen.

Programming Considerations:

-- If the literal is an octal control character (see section 2.5), it is sent to the terminal.

-- If the literal is of the form "<string>", the following rules

apply.

a) All of the characters between the double quotes are displayed as they appear in the literal.

b) The first character of the string is displayed at the current cursor position.

c) The cursor is bumped one position to the right for every character displayed.

d) The cursor is left positioned one position to the right of the last character of the literal.

9.3 CONSOLE

The CONSOLE instruction is used to put information on the console screen. This statement has the following general format:

<label> CONSOLE <list>

where: <label> is an execution label (see section 2.).
<list> is a list of items describing the information to be put on the console.

Programming Considerations:

-- <label> is optional.

-- The items in the list must be separated by commas.

-- <list> may be made up of any combination of the following items:

a) <svar> is a character string variable (see section 4.2).

b) <nvar> is a numeric string variable (see section 4.1).

c) <occ> is an octal control character (see section 2.5).

d) <list control> is used to control the manner in which the list is processed.

e) <slit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid character string (see section 4.2).

f) <nlit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid numeric string (see section 4.1).

-- All output to the system console is inhibited if it is being used as the terminal for port one. In this case, all CONSOLE instructions execute, but do not actually do anything.

-- The output always is on the line assigned for the terminal executing the CONSOLE instruction. This means that any vertical positioning of the cursor is ignored.

-- A CONSOLE statement which begins without positioning will start displaying at column 5.

-- The port number and asterisk appearing in column 1 through 4 on the CONSOLE may be overwritten by positioning to column 1.

-- Character string variables are handled exactly alike in CONSOLE and DISPLAY statements (for more details, see section 9.2.1).

-- Numeric string variables are handled exactly alike in CONSOLE and DISPLAY statements (for more details, see section 9.2.2).

-- The only DISPLAY list control that is effective is *P<h>:<v> (cursor positioning). The cursor positioning used for CONSOLE statements works like it does for KEYIN statements except that the vertical position (<v>) is ignored (for more details, see section 9.1.3.1).

-- If the display flows over the line length 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 are ignored.

Example: The CONSOLE instruction could be used to alert the system operator (if such a person exists) by using the following statement.

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

9.4 BEEP

BEEP causes a beep (ASCII "ring bell" character) to be sounded at the terminal. This instruction has the following general format:

```
<label>  BEEP
```

where: <label> is an execution label (see section 2.). This label is optional.

9.5 DEBUG

The DEBUG instruction is used to activate the interpreter's debugging tool, if such a tool exists. The user's guide of the appropriate interpreter should be consulted for details on the operation of this tool. This instruction has the following general format:

```
<label>  DEBUG
```

where: <label> is an execution label (see section 2.).

Programming Considerations:

-- <label> is optional.

-- If the debugging tool is not available, DEBUG is treated like "No OPeration" (NOP). That is; program execution continues as if the DEBUG instruction had not been included in the program.

CHAPTER 10. PRINTER OUTPUT

These instructions are used to output data to a printer and to control the usage of the printer by a port.

General Programming Considerations:

-- Typically, formatting is handled in one of the following ways.

   a) By the way the variable is defined. It should be defined with the format which is to be used for output.

   b) Using list controls.

-- Normally, when execution of PRINT (or RPRINT) statement terminates, the print position is reset to the beginning of the next line.

-- If a semicolon (;) is used after the last item in the list, the print position remains where it was on statement termination. This feature allows a second PRINT (or RPRINT) statement to continue where the first statement left off.

   Example:

```
                PRINT     "FLAGS: ";
                CALL      NOTFLG IF NOT ZERO
                PRINT     "ZERO, ";
                CALL      NOTFLG IF NOT LESS
                PRINT     "LESS"
                ...
        NOTFLG  PRINT     "NOT ";
                RETURN
```

   would print one of the following lines, depending on the condition flags.

```
FLAGS: ZERO, LESS
FLAGS: ZERO, NOT LESS
FLAGS: NOT ZERO, LESS
FLAGS: NOT ZERO, NOT LESS
```

-- Those instructions that use a list should make use of continuation when it is possible to do so. (For details about using continuation, see section 2.) This not only increases

the execution speed of the program, but also decreases the system overhead. The programmer should check his program for any occurance of two consecutive PRINT statements to see if they can be combined into a single statement.

```
PRINT     "LINE ONE"
PRINT     "LINE TWO"
```

should be combined to form the statment below.

```
PRINT     "LINE ONE":
          *N,"LINE TWO"
```

10.1 PRINT

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 format of the print instruction is:

1) <label> PRINT <list>

Where: <label> is an execution label.
<list> is a list of items describing the output to the printer.

Programming Considerations:

-- <label> is optional.

-- The items in the list must be separated by commas.

-- <list> may be made up of any combination of the following items:

a) <svar> is a character string variable (see section 4.2).

b) <nvar> is a numeric string variable (see section 4.1).

c) <occ> is a octal control character (see section 2.5).

d) <list control> is used to control the manner in which the printing is performed.

e) <slit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid character string (see section 4.2).

f) <nlit> is a literal of the form "<string>" (see section 2.5). <string> must be a valid numeric string (see section 4.1).

10.1.1 Character String Variables

When a character string variable (<svar>) appears in the list of a PRINT (or RPRINT) instruction, the characters stored in the variable are printed on the printer. Unless modified by a list control the manner in which the characters printed on the printer is described below.

Programming Considerations:

-- The characters in the variable are printed starting with the first physical character and continuing through the logical length.

-- Blanks will be printed for any character positions that exist between the logical length pointer and the physical end of the variable.

-- The first character printed will be printed at the current print position.

-- The print position will be incremented by one for each character printed.

-- The current print position is left positioned one character to the right of the last character printed.

10.1.2 Numeric String Variables

When a numeric string variable (<nvar>) appears in the list of a PRINT instruction, the characters that are stored in the variable are printed on the printer. Unless modified by a list control, the manner in which the characters are printed is described below.

Programming Considerations:

-- The characters printed start with the first physical character and continue through the physical end of the variable.

-- The first character printed will be printed at the current

print position.

-- The print position is left positioned one character to the right of the last character printed.

10.1.3 List Controls

The list controls are provided to allow more flexibility in the way the printer is formatted. They may be used to control the manner in which variables are printed on the printer. All list controls begin with an asterisk followed by the specification of the control function.

10.1.3.1 *F (Form Feed)

The *F control causes the printer to advance to the top of the next form and the print position to be set to the first column.

10.1.3.2 *C (Carriage Return)

The *C control causes the print position to be set to the beginning of the current line.

10.1.3.3 *L (Line Feed)

The *L control causes the print position to be set to the following line in the current print position. For example: if the print position was column 20, then the *L control would cause the print position to be unchanged on the following print line.

10.1.3.4 *N (Next Line)

The *N control causes the print position to be set to one (1) on the following line. Executing the *N control is equivalent to executing a *C control followed by a *L control.

10.1.3.5 *<n> (Tab To Column <n>)

The *<n> control causes the print position to be set to column (n). <n> must be an integer constant.

10.1.3.6 ; (Supress new line function)

The (;) control causes the new line function to be supressesed. This control inhibits the *N control function which normally occurs at the end of a PRINT instruction without the (;) control.

10.1.3.7 *ZF (Zero Fill)

The *ZF control may be used before a numeric variable to cause zero fill on the left, moving the sign to the left if necessary.

10.1.3.8 *+ (Blank Supression On)

This control is used to turn on a print mode called blank suppression.

Programming Considerations:

-- This control affects printing of all character string variables which follow the *+ control in the PRINT list.

-- If blank suppression is turned on, character string variables are printed on the printer as described below.

   a) The characters in the variable are printed starting with the first physical character and continuing through the logical length.

   b) The first character printed will be printed at the current print position.

   c) The current print position is incremented by one (1) for each character printed.

   d) The print position is left positioned one character to the right of the last character printed.

-- Blank suppression is automatically turned off when the end of

the PRINT list is reached.

10.1.3.9 *- (Blank Suppression Off)

This control turns blank suppression mode off. For more details about blank suppression mode, see section 10.1.3.8.

10.1.4 Literals

When a literal (<occ>, <slit> or <nlit>) appears in the list of a PRINT (or RPRINT) statement, that literal is printed on the printer.

Programming Considerations:

-- If the literal is an octal control character (see section 2.5), it is sent to the printer.

-- If the literal is of the form "<string>", the following rules apply.

   a) All of the characters between the double quotes are printed as they appear in the literal.

   b) The first character of the string is printed at the current print position.

   c) The print position is incremented one position to the right for every character printed.

   d) The print position is left positioned one position to the right of the last character in the literal.

10.2 RPRINT

The RPRINT instruction functions exactly as the PRINT except that the printout physically occurs at a Remote Slave Station instead of the Central Station where the PRINT instruction functions. The format of the RPRINT instruction is:

```
<label>  RPRINT    <list>
```

Where: <label> is an execution label.
<list> is a list of items describing the output to the

printer.

Programming considerations:

-- <label> is optional.

-- If the port is not a remote slave port type, then the instruction is interpreted as a PRINT instruction.

The user should refer to section 10.1 for a discussion of the PRINT statement.

10.3 RELEASE

The RELEASE instruction ends a user's (port's) exclusive control of the printer and causes the printer to advance to the top of the next form. The instruction has the following format:

<label> RELEASE

Where: <label> is an execution label.

Programming Considerations:

-- <label> is optional.

-- This instruction causes the printer to become available to another user.

-- The printer is seized by a user when the user attempts to perform a PRINT instruction and the printer is not in use by another port.

-- The printer will advance to the top of the next form.

-- When the user disconnects from the system or keys the interrupt procedure on the keyboard, a RELEASE is automatically performed for that user.

-- This instruction has no effect upon printing being performed at the remote slave station.

-- This instruction is ignored on non-DATASHARE Systems.

10.4 Printer Considerations

The tabbing (*<n>) in the PRINT (or RPRINT) statement can move the carriage in the reverse direction and any sequence of printer controls will be executed in precisely the sequence specified.

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

Another feature allowed with the servo printer is minor vertical spacing. The following list depicts the octal control characters (OCC) which are used for the vertical minor spacing and the horizontal column spacing. There are eight (8) minor vertical spaces for one standard line space.

OCC	FUNCTION
000	Vertical minor spacing 0 spaces (down the page)
001	Vertical minor spacing 1 space (down the page)
002	Vertical minor spacing 2 spaces (down the page)
003	Vertical minor spacing 3 spaces (down the page)
004	Vertical minor spacing 4 spaces (down the page)
005	Vertical minor spacing 5 spaces (down the page)
006	Vertical minor spacing 6 spaces (down the page)
007	Vertical minor spacing 7 spaces (down the page)
010	Vertical minor spacing 0 spaces (up the page)
011	Vertical minor spacing 1 space (up the page)
012	Vertical minor spacing 2 spaces (up the page)
013	Vertical minor spacing 3 spaces (up the page)
014	Vertical minor spacing 4 spaces (up the page)
015	Vertical minor spacing 5 spaces (up the page)
016	Vertical minor spacing 6 spaces (up the page)
017	Vertical minor spacing 7 spaces (up the page)
020	Left carriage movement 7 columns
021	Left carriage movement 6 columns
022	Left carriage movement 5 columns

023	Left carriage movement 4 columns
024	Left carriage movement 3 columns
025	Left carriage movement 2 columns
026	Left carriage movement 1 column
027	No action
030	Right carriage movement 1 column
031	Right carriage movement 2 columns
032	Right carriage movement 3 columns
033	Right carriage movement 4 columns
034	Right carriage movement 5 columns
035	Right carriage movement 6 columns
036	Right carriage movement 7 columns
037	Right carriage movement 8 columns

These features on the servo printer allows different kinds of underscoring and super- and/or sub-scripting in the printed output. Note that it is the user's responsibility to keep track of the carriage micro-position.

CHAPTER 11. COMMUNICATIONS INPUT/OUTPUT

The following instructions are used for communications between ports (internal communications) and for communications to a remote or host computer (external communications).

11.1 SEND

The SEND instruction is used to transmit a list of data variables to a specified destination. The statement has the following format:

<label> SEND <cmlst>,<route>;<nslst>

where: <label> is an execution label.
<cmlst> is a variable with the COMLST data declaration.
<route> is a string variable that contains the routing information for the list of variables.
<nslst> is a list of variables either numeric or character string that are to be transmitted to the specified destination.

Programming Considerations:

-- <label> is optional.

-- <cmlst> must be a variable with the COMLST data declaration.

-- <route> must be a string variable. The formpointed character in the string must be either an "I" specifying internal communications (between ports) or an "E" specifying external communications (with a remote or host computer).

-- For internal communications (between ports), the two characters following the "I" must be valid numeric digits and are used as the destination port for the data contained in the list <nslst>.

a) A port number of "01" is port 01 or the first port in the system.

b) If there are not two valid numeric digits after the formpointed character in the <route> variable, then the <cmlst> variable is set to 'clear'. An I/O trap is given

and the rest of the instruction is ignored. The SEND operation is not performed.

c) If the destination port is not configured into the system, then the 'channel unavailable' status is set into the <cmlist>. The SEND operation is not performed.

d) If a RECV operation is 'pending' on the destination port the data from the variable(s) in the <nslst> are transferred to the variable(s) specified in the RECV instruction at the destination port. The data is transferred on a variable to variable basis. That is the first variable in the SEND statement is transferred to the first variable in the RECV statement and the second variable (SEND) in the second variable (RECV) until either the SEND or RECV list is exhausted. If a SEND variable is longer than the RECV variable, then the excess data is discarded.

e) If no RECV operations are 'pending' at the destination port, then the <cmlst> status is set to 'channel unavailable', and the instruction is ignored. The SEND operation is not performed.

f) For character string variables, the data is transmitted starting with the first physical character through the logical length.

g) For numeric variables, the data is transmitted starting with the first through the last physical character.

h) The internal SEND operations are performed in the background.

-- For external communications (to/from a host or remote computer) the following considerations are pertinent:

a) The information after the "E" in the <route> variable is a function of the communications process being used. The compatible line handler user's guide should be consulted for this information.

b) If the external communications has not been configured into the system or is not available the instruction is ignored and the <cmlst> status is set to 'channel unavailable'.

c) If the external communications is available then the SEND

instruction is processed and the status of the <cmlist> is set to 'pending' and the next DATAUS instruction is executed. The data may not be transferred to the remote or host computer immediately, therefore the DATABUS programmer must not modify any of the variables mentioned in the SEND statement until the status of the <cmlst> indicates that the SEND is complete.

d) The data transmitted for external communications is from the first physical character through the logical length. Consult the communication line handler user's guide for details.

-- If the routing variable <route> formpointed character contains neither an "E" or "I" then the instruction is ignored and an I/O trap is given.

Example:

```
CMLST   COMLST    5
ROUTE   INIT      "I16"
VAR1    INIT      "MESSAGE NUMBER"
VAR2    FORM      4
VAR3    INIT      "THIS IS YOUR MESSAGE"
        .
        .
WAIT    COMCLR    CMLST
        SEND      CMLST,ROUTE;VAR2,VAR2:(SEND MESSAGE)
                  VAR3
        COMTST    CMLST                 (GET CMLST STATUS)
        GOTO      WAIT IF OVER          (DESTINATION PORT NOT
                                        READY)
        DISPLAY   "MESSAGE NUMBER",VAR2,"TRANSFERRED OK"
        .
        .
        .
        .
```

11.2 RECV

The RECV instruction is used to specify a list of variables which will serve as a destination for data from a source. The statement has the following format:

```
<label>  RECV      <cmlst>,<route>;<slist>
```

where: <label> is an execution label.

<cmlst> is a variable with the COMLST data declaration.
<route> is a string variable which contains routing information.
<slist> is a list of string variables which are to receive the data.

Programming Considerations:

-- <label> is optional.

-- <cmlst> is a variable with the COMLST data declaration.

-- <route> is a string variable which contains the routing information. An "I" specifies internal communication (between ports) and an "E" specifies external communication (with a host or remote computer).

-- For internal communications, the following facts are pertinent:

a) There must exist two valid numeric characters after the formpointed (after the "I") character in the <route> variable. These two numeric characters specify the port that is expected to SEND the data. If the expected SENDing port is invalid then an I/O error is given and the rest of the instruction is ignored. If the expected SENDing port is not configured then the <cmlst> status is set to 'channel unavailable' and the rest of the instruction is ignored. The actual SENDing port and the expected SENDing port numbers may be different. The expected SENDing port is used to simply tell the DATABUS program if the port is configured.

b) When data is received from another port, then the two characters following the formpointed character (the "I") in the <route> variable are overstored with the port number that originated the data (SENDing port number). A port number of "01" specifies the port 1 was the SENDing port.

c) If there is no SEND operation 'pending' on another port with the destination for the RECVing port, then the <cmlst> status is set to 'communications pending' or 'in process'. Otherwise the RECV operation occurs immediately.

d) The data is transferred from the SENDing to the RECVing port on a variable to variable basis. That is the first

SENDing variable is stored into the first RECVing variable and the second SENDing variable to the second RECVing variable until either the SENDing or RECVing list is exhausted.

e) If a RECVing variable will not contain all of the data for the SENDing variable, then the excess data is discarded.

f) If the SENDing variable list contains more variables than the RECVing variable list, the excess variables are discarded.

g) If the SENDing variable list contains fewer variables than the RECVing variable list, the excess variables that did not receive data has their formpointer and logical length set to zero.

h) The logical length of the RECVing variables reflect the amount of data transferred. The formpointer is reset to 1.

-- For external communications, the following facts are pertinent:

a) If the external communications has not been configured into the system, or is not available then the instruction is ignored and the <cmlst> status is set to 'channel unavailable'. The next DATABUS instruction is executed.

b) The logical length is set on all RECVing variables to reflect the quantity of data received for the variable. The formpointer is reset to 1.

c) The communication line handler user's guide should be consulted for additional details on external RECV operation.

-- If the formpointed character in the <route> variable contains neither an "E" nor "I" then the rest of the instruction is ignored and an I/O trap is given.

Example:

```
CMLIST   COMLST    1
CMLIST1  COMLST    3
ROUTE    INIT      "I08"
ROUTE1   INIT      "I08"
```

```
VAR1      INIT      "PLEASE SEND ME YOUR TIME REPORTS"
EMPLN     DIM       5
DATE      DIM       10
HOURS     DIM       3
          .
          .

          SEND      CMLIST,ROUTE;VAR1     (SEND THE MESSAGE)
TEST      COMTST    CMLIST                (TEST THE CMLIST)
          GOTO      TEST IF LESS          (SEND NOT COMPLETE)
          GOTO      NOTAVAL IF OVER       (CHANNEL NOT AVAILBLE)
CYCLE     COMCLR    CMLIST1               (CLEAR THE COMLIST)
          RECV      CMLIST1,ROUTE1;EMPLN,DATE,HOURS
TEST1     COMTST    CMLIST1               (RECV COMPLETE)
          GOTO      NOTAVAL IF OVER       (CHANNEL UNAVAILABLE)
          GOTO      TEST1 IF LESS         (RECV NOT COMPLETE)
          .
          .
          .         (STORE DATA)
          .
          .
          GOTO      CYCLE                 (GET MORE DATA)
NOTAVAL   DISPLAY   "CHANNEL UNAVAILABLE"
          .
          .
          .
```

11.3 COMCLR

The COMCLR instruction is used to clear the status of the specified communications list <COMLST>. The instruction has the following format:

```
<label>  COMCLR    <cmlst>
```

where: <label> is an execution label.
<cmlst> is a variable with the COMLST data declaration.

Programming Considerations:

-- <label> is optional.

-- <cmlst> must be a variable with the COMLST data delaration.

-- If the actual status of the <cmlist> is 'pending' or 'in process', and a message is being transferred, then the message being transferred will be truncated.

-- If a <cmlst> appears in a SEND or RECV statement, it may not appear in another such statement without first appearing in an intervening COMCLR statement.

-- The <cmlst> status is set to 'clear' when this instruction is executed.

Example:

```
CLIST   COMLST    5
ROUTE1  INIT      "I03"
ROUTE2  INIT      "I15"
MSG     INIT      "PLEASE NOTIFY EMPLOYEES OF MEETING TODAY"
          .
          .
TEST    COMCLR    CLIST                    (CLEAR COMLIST)
        SEND      CLIST,ROUTE1;MSG         (SEND MESSAGE)
        COMTST    CLIST                    (SEND COMPLETE?)
        GOTO      TEST IF LESS             (RETRY SEND)
        GOTO      TEST IF OVER             (RECV PORT NOT
                                           READY.)
                                           (SEND COMPLETE)
NEXT    COMCLR    CLIST                    (CLEAR THE COMLIST
                                           FOR REUSE)
        SEND      CLIST,ROUTE2;MSG         (NEXT SEND
                                           OPERATION)
          .
          .
          .
          .
```

11.4 COMTST

The COMTST instruction is used to access the status information stored in the communications list <CMLST> The COMTST instruction has the following format:

```
<label>  COMTST    <cmlst>
```

where: <label> is an execution label.
<cmlst> is a variable with the COMLST data declaration.

Programming Considerations:

-- <label> is optional.

-- <cmlst> must be a variable with the COMLST data declaration.

-- After the COMTST instruction is executed the flags are set as follows:

EQUAL - Communication completed successfully.

OVER - 'Channel unavailable'. For internal communications (communications between ports) this means that the port specified to receive the data is not configured into the system. For external communications (communication with a remote or host computer) this means that either the external communications was not configured or was not available.

LESS - 'Communications pending' or 'in process'. This means that the none of the variables specified in the SEND or RECV instructions should be modified before a subsequent COMTST instruction yields an EQUAL condition signifying that the process is complete.

-- If all three of the above conditions (less, over, equal) are false, the <cmlst> variable is said to be 'clear' which means that it is free to be used in a SEND or RECV statement.

Example:

```
CMLIST    COMLST   5
ROUTE     INIT     "I05"
V1        INIT     "THIS IS YOUR MESSAGE"
V2        DIM      50
          .
          .
WAIT      COMCLR   CMLIST
          SEND     CMLIST,ROUTE;V1,V2
          COMTST   CMLIST                (GET STATUS OF CMLIST)
          GOTO     WAIT IF OVER          (DESTINATION PORT NOT
                                         READY TO RECV)
NXTMSG    .                              (PROCEED WITH NEXT
                                         MESSAGE)
          .
          .
          .
```

11.5 COMWAIT

The COMWAIT instruction is used to suspend program execution at a DATASHARE port. Execution is suspended until either a SEND or a RECV instruction (see sections 11.1 and 11.2) indicates I/O completion. This instruction has the following format:

```
<label>  COMWAIT
```

where: <label> is an execution label (see section 2.).

Programming Considerations:

-- <label> is optional.

-- If no SEND or RECV instructions have initiated communication, the COMWAIT instruction is treated like a "No OPeration" (NOP) instruction. That is; execution continues with the next instruction as if the COMWAIT instruction had not been included in the program.

-- If any communications ('pending' or 'in process') are active when the COMWAIT instruction is executed, then execution of the program is suspended. That is; program execution does not continue with the next instruction until a signal to continue is received. This suspension of program execution imposes very litle overhead on a DATASHARE system.

-- Termination of any one of the communication processes indicates to the COMWAIT instruction that it should resume execution. This allows the programmer to avoid putting the COMTST (see section 11.4) within a tight loop to check for termination of a communication task. Such tight loops impose considerable overhead on a DATASHARE system.

-- Since any communication process may cause execution to resume, a series of COMTST instructions must be used to determine which process terminated. This series of tests imposes much less overhead on the system than the tight loop method described above.

Example:

```
A         COMLST    3
B         COMLST    5
AROUTE    INIT      "E00"
BROUTE    INIT      "E00"
AVAR      INIT      "STRING1"
BVAR      INIT      "STRING2"
          SEND      A,AROUTE;AVAR
          SEND      B,BROUTE;BVAR
WAIT      COMWAIT
          COMTST    A
          GOTO      ACOMP IF EQUAL
          COMTST    B
          GOTO      BCOMP IF EQUAL
          ...
ACOMP     COMCLR    A
          ...
          (Modify AVAR)
          ...
          SEND      A,AROUTE;AVAR
          GOTO      WAIT
BCOMP     COMCLR    B
          ...
          (Modify BVAR)
          ...
          SEND      B,BROUTE;BVAR
          GOTO      WAIT
```

CHAPTER 12. DISK INPUT/OUTPUT

These instructions make use of the Datapoint DOS file structure while reading from and writing to the disk. For more details about this structure, see the DOS User's Guide and the Systems Guide of the appropriate DOS. Basically, the DOS file structure is as follows.

The smallest unit of storage on the disk is the sector. That is; all disk I/O hardware operations affect entire sectors, never a partial sector. Each sector is capable of saving up to 251 bytes of information (there are actually 256 bytes per sector, but 5 bytes are reserved for use by DOS).

In most cases, the information to be saved will not fit within one sector. To handle such information, sectors are arranged into groups called files.

The DOS file structure is made up of files arranged so that they can be easily referenced by names associated with them. (The name associated with a file is usually selected by the user.)

A good analogy is to think of the DOS file structure as follows:

```
file structure   = file cabinet
file             = folder in the cabinet
sector           = sheet of paper in the folder
```

This analogy will be used later in the discussion of disk I/O.

Note that; the disk structures on the remote station disks (diskettes) and the central station disks are identical from the programmers point of view. The only difference depends on whether the file was declared using RFILE or RIFILE, rather than FILE or IFILE. If it was declared using RFILE or RIFILE, the file accessed will be on a remote station disk (diskette). If it was declared using FILE or IFILE, the file accessed will be on a central station disk.

12.1 File Structure

When a group of sectors is organized into a file; some information, about the location of those sectors, must be kept by DOS and the Databus interpreter.

Databus keeps its information about each file in the user's data area. The file declaration statements (see Chapter 5.) are used to reserve space in the user's data area for this information.

The information kept by Databus is described below.

-- The drive number of the disk drive on which the file is found.

-- A pointer to the physical location of the file.

-- The following pointers which describe the current position within the file.

   a. The record number, which points to the sector currently being referenced. A record number of 0 indicates the first sector within the file.

   b. The character pointer, which points to the byte currently being referenced within the sector. The first byte of the sector is indicated by the character pointer being equal to 1.

-- A counter used to keep track of the number of spaces when using space compression (for more details on space compression, see section 12.1.2).

-- Two additional pointers are included for use with index files only. These are:

   a. A pointer to the logical record last referenced by using the index file.

   b. A pointer to the next key in sequence. (All of the keys in the index file are sorted using their ASCII values.)

12.1.1 Record Structures

There are several ways of organizing records on the disk sectors. All of them provide different methods of accessing the information saved on the disk. The types of records that can be used are physical records, logical records and indexed records.

12.1.1.1 Physical Records

Programming Considerations:

-- A physical record corresponds to exactly one sector on the disk.

-- A physical record starts with the first character of the sector.

-- An 003 (octal) character terminates a physical record.

-- There are at most 250 data characters in a physical record. (Note: when considering physical records the logical end-of-record character, 015, is treated as a data character.)

Analogy:

```
file structure   = file cabinet
file             = folder in the cabinet
sector           = sheet of paper in the folder
physical record  = page of text on the sheet of paper
```

12.1.1.2 Logical Records

Programming Considerations:

-- A logical record is terminated with an 015 (octal) character.

-- A logical record starts with the character immediately following the 015 of a previous logical record.

-- More than one logical record may be saved on a physical record.

-- Logical records may extend across physical record boundaries.

-- There is no restriction upon the length of a logical record. A single logical record may extend across many physical

records. (It is a good idea to keep logical records reasonably short to make them easy to deal with.)

Analogy:

file structure	= file cabinet
file	= folder in the cabinet
sector	= sheet of paper in the folder
physical record	= page of text on the sheet of paper
logical record	= paragraph of text on the sheet of paper

Example: Four logical records could appear on the disk as follows:

```
asc asc asc asc asc asc oct asc asc asc asc asc asc oct asc oct
 L   I   N   E       1  015  L   I   N   E       2  015  L  003

asc asc asc asc asc oct asc asc asc asc asc asc oct oct
 I   N   E       3  015  L   I   N   E       4  015 003
```

Note that the first physical record contains two logical records as well as the first letter of a third. The third logical record starts in the first physical record and continues into the second physical record. At this point the fourth logical record starts and continues to the end of the physical record.

Example: If the same four logical records were written to the disk one per physical record, they would appear as follows:

```
asc asc asc asc asc asc oct oct
 L   I   N   E       1  015 003

asc asc asc asc asc asc oct oct
 L   I   N   E       2  015 003

asc asc asc asc asc asc oct oct
 L   I   N   E       3  015 003

asc asc asc asc asc asc oct oct
 L   I   N   E       4  015 003
```

Note that it took twice as much disk space to save the same amount of information in this example than in the previous example. It

is sometimes desirable to give up this disk space to provide faster and easier access to a logical record.

12.1.1.3 Indexed Records

An indexed record is a logical record that is named. This makes it possible to reference a record by simply specifying the name of the record.

Programming Considerations:

-- The name that is associated with the logical record is called a key.

-- There is no distinction between a data file that is indexed and one that is not.

-- All of the keys, associated with the records in a data file, are saved in a separate file. This file, that contains the keys for another data file, is called an index file.

-- There may be more than one index file associated with a single data file.

-- Databus requires that all index files have the DOS file extension of /ISI.

-- The index file contains:

   a. The name and extension of the data file which it indexes.

   b. The keys.

   c. The pointers necessary to associate the keys with the logical records.

-- The DOS INDEX command is the only way that index files can be created. For more details on INDEX, see the DOS User's Guide.

-- All keys put into the index file by the DOS INDEX utility will not have any trailing spaces. (Unnecessary spaces cause larger index files and longer access times.)

-- The index structure is an n-ary tree, where:

   a. n is determined by the number of keys that will fit within a sector.

b. Each node of the tree is contained within one disk sector.

c. The tree has enough levels so that the uppermost node will fit within one disk sector.

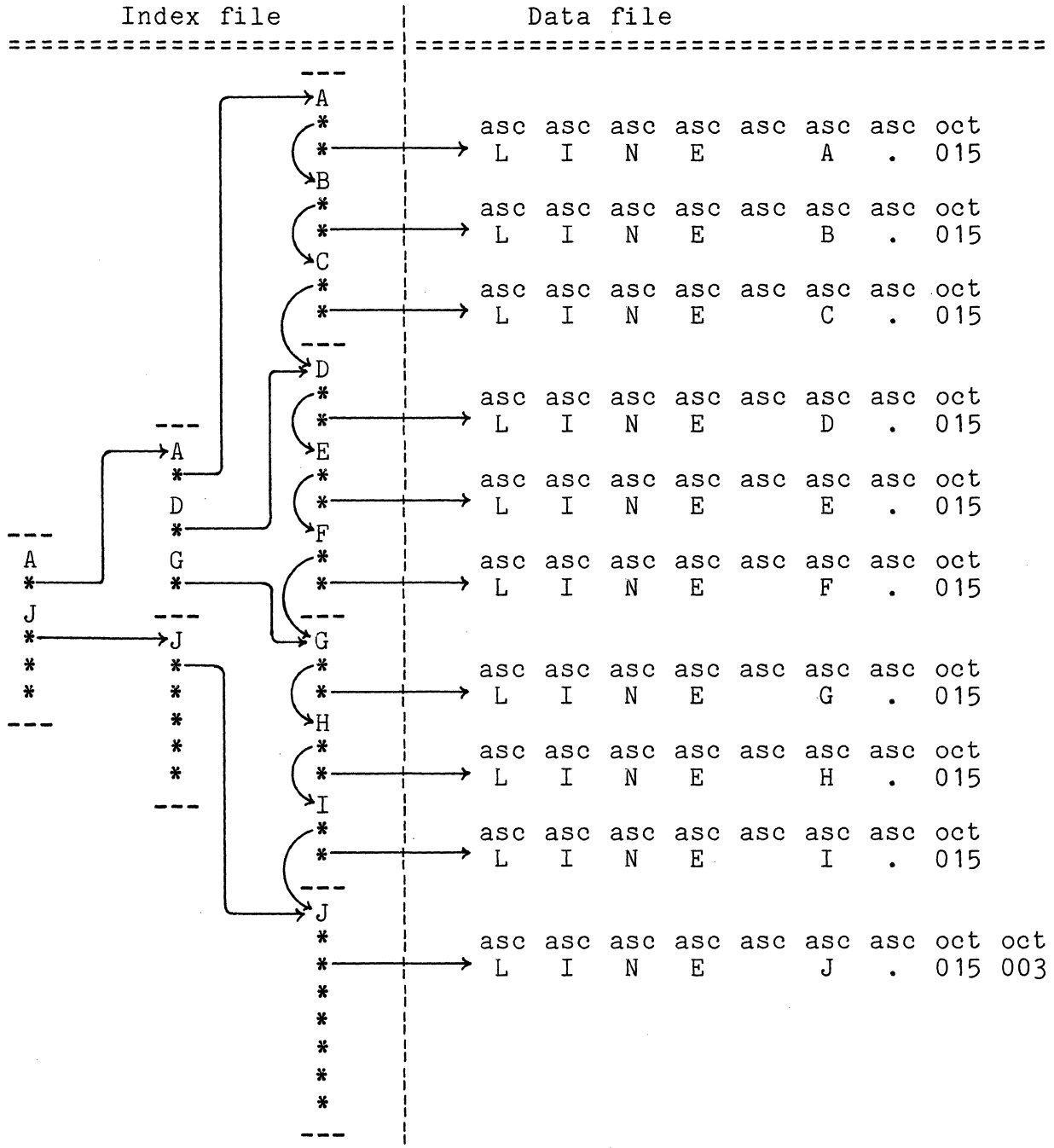
-- The lowest level of the tree is a linked list. The keys in the linked list are arranged sequentially according to their ASCII values.

-- Depending on the length and path of this linked list, the time spent in traversing this list can lead to considerable overhead. The INDEX utility may be used to reorganize this list to minimize the time spent in traversing it. USE THE INDEX UTILITY FREQUENTLY!

Analogy:

file structure	= file cabinet
file	= folder in the cabinet
index file	= folder that contains the table of contents of another folder
sector	= sheet of paper in the folder
physical record	= page of text on the sheet of paper
logical record	= paragraph of text on the sheet of paper

The following diagram demonstrates the way in which the keys are associated with the logical records. The diagram assumes that only 3 keys will fit per sector and that the data file was indexed on column 6. The *'s indicate pointers. Sector boundaries are indicated by ---.



12.1.2 Space Compression

In some data files large numbers of contiguous spaces appear. The disk space used by such files may be compressed by replacing the contiguous spaces with a count of the spaces. The following programs all produce space compressed disk files: EDIT, SORT, REFORMAT, DBCMP (print files), several terminal emulators and all of the Databus interpreters.

Space compression is done by counting the contiguous spaces, then replacing them with the following: the 011 (octal) control character followed by a byte which contains the count of the spaces. This number will never be less than 2 (since it is wasteful to expand one or zero spaces into two characters) and may be as large as 255. Any program that encounters the 011 on the disk will then look at the next byte to get the number of spaces that should appear at that point in the record. The 011 will never appear as the last character in a physical record. This prevents the 003 (end of physical record) from being used as a count of 3 spaces.

Trailing spaces are never written to space compressed records. The 015 (end of logical record) character is always written immediately after the last non-blank character in the record.

If the record is to be modified in place, using space compression is discouraged. If the number of spaces is changed by the modification, the position of any non-blank characters may be shifted within the physical record. This could easily cause a FORMAT trap on subsequent reads from that record.

Example: The following a logical record is shown first without space compression and then with space compression.

```
asc asc asc asc asc asc asc asc asc asc asc asc asc oct oct
 1       2           5                       X          015 003

asc asc asc oct oct asc oct oct asc oct oct
 1       2  011 002  5  011 005  X  015 003
```

12.1.3 End of File Mark

The end of file mark (EOF) is a special type of physical record which is written to the disk as the last physical record of a file.

The end of file mark always starts at the beginning of a physical record and looks like the following physical record:

```
oct oct oct oct oct oct oct
000 000 000 000 000 000 003
```

The rest of the characters in the sector are of no significance.

All records between the beginning of the file and the EOF must be in acceptable physical record format. Any record that is not in this format will cause an I/O or FORMAT trap. An empty file is acceptable; that is, any file which has an EOF as its first physical record is acceptable.

12.2 Accessing Methods

All disk I/O in Databus is based upon establishing a position within a file. Once this position is established, all accesses are performed by moving this position within the file. This position within the file is completely described by the record number and character pointer mentioned in section 12.1.

Bumping the postion in a file refers to bumping the character pointer; with the following exception. If the character pointer is bumped to the physical end-of-record character (003), the following actions are taken:

a. the record number is bumped by one, and

b. the character pointer is set to one.

12.2.1 Physical Record Accessing

Physical record accessing is the fastest and simplest method of accessing information within a file. Physical record accessing may be used to randomly access information on the disk.

Programming Considerations:

-- Each physical record in a file is assigned a positive integer

number. 0 is assigned to the first physical record in the file, 1 to the second, 2 to the third, and so on to the last record in the file.

-- To access a record, the programmer must specify the record number of the physical record he wishes to use.

-- The position in the file is modified to be:

   a. The record number of the file is set to the number supplied by the programmer.

   b. The character pointer is set to one.

-- Once the position has been established, the access continues as if it had been a logical record access (see section 12.2.2).

12.2.2 Logical Record Accessing

This is the access method used to read and write logical records. This access method allows only sequential processing of disk records. If random access to logical records is desired, the slower indexed accessing must be used.

Programming Considerations:

-- The position within the file is not reset initially.

-- The position within the file is bumped by one for every character accessed on the disk.

-- Bumping the position to the physical end-of-record character is described in section 12.2.

-- When the logical end-of-record character (015) is read/written, the following actions are taken:

   a. record processing is terminated.

   b. the position within the file is bumped past the 015.

12.2.3 Indexed Record Accessing

This method is used to reference logical records randomly rather than sequentially. While this method provides greater flexibility in random accessing, it is also much slower. If the time spent in accessing the disk is critical, a means of using physical record accessing should be used.

Programming Considerations:

-- There are five basic indexed operations:

   a. Read the named logical record.

   b. Read the next record in sequence. (The keys are sorted using the ASCII values of the keys.)

   c. Insert the named logical record.

   d. Delete the named logical record.

   e. Modify the named logical record.

-- Since there can be any number of indexes into one data file, inserting (or deleting) a record involves inserting (or deleting) the key into (or from) all of the indexes.

-- In addition to the position within the data file, DATABUS maintains another position within the index file. Once this position has been established it is used to access the record whose key is next in the collating sequence (the key next in alphabetical order).

-- The position within the data file is established by finding the key in the index file and using the pointers saved there as the position. This does not apply to insertions, since the key is not in the index file yet.

-- The position within the data file for insertions is always at the end of the data file. For more details, see section 12.2.

-- Once the position within the data file has been established, the access continues as if it had been a logical record access (see section 12.2.2).

-- An indexed access will cause the following number of disk sectors to be read.

a. One sector for each level of the index except the lowest level.

b. At least one sector for the lowest level of the index. The number of disk reads at this level can become very large, if the index file has not been re-built recently. This is particularly true if a large number of keys have been inserted into the index. USE THE INDEX UTILITY FREQUENTLY!

c. Whatever disk functions are required to perform the actual read or write operation.

-- The linked list at the lowest level of the index will have a very long and disorganized path when a data base is initialized using insertions. This leads to considerable overhead. If a data base must be initialized using insertions, using the INDEX utility to clean-up the index is particularly important.

-- Both physical record and logical record accesses can be made to indexed files.

12.3 General Instructions (Disk I/O)

There are many aspects of some of the Disk I/O instructions which are common to all of the acessing methods. The following sections discuss these common aspects of several of the instructions.

12.3.1 OPEN (General)

This instruction is used to initialize a logical file for use by a Databus program. The use of logical files allows a Databus label to be associated with a file on the disk. One of the following general formats may be used:

```
1)   <label>  OPEN     <file>,<slit>
2)   <label>  OPEN     <file>,<svar>
3)   <label>  OPEN     <ifile>,<slit>
4)   <label>  OPEN     <ifile>,<svar>
5)   <label>  OPEN     <rfile>,<slit>
6)   <label>  OPEN     <rfile>,<svar>
7)   <label>  OPEN     <rifile>,<slit>
8)   <label>  OPEN     <rifile>,<svar>
```

where: <label> is an execution label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<svar> is a string variable (see section 4.2).
<file> is a file declared using the FILE declaration (see section 5.1).
<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rfile> is a file declared using the RFILE declaration (see section 5.3).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- The value of <svar> is unchanged by the execution of this instruction.

-- The string literal, when using format (1), (3), (5) or (7); specifies the DOS name of the disk file to be associated with the label.

-- The string variable, when using format (2), (4), (6) or (8); specifies the DOS name of the disk file to be associated with the label.

-- The extension is not furnished by the string literal or string variable. The following extensions are assumed:

   a) /TXT for those files opened using formats (1), (2), (5) and (6).

   b) /ISI for those files opened using formats (3), (4), (7) and (8).

-- One of the following rules is used to build the DOS name from the string in the string variable or string literal:

   a) The characters used start with the formpointed character and continue until eight characters have been obtained, or

   b) if the logical end of string is reached before eight characters have been obtained, the remainder of the eight

characters are assumed to be blanks.

-- The character used to specify the drive number is obtained from the string variable or string literal using one of the following rules:

   a) If (a) above is used to obtain the name, then the character after the eighth character is used as the drive specification, or

   b) If (b) above is used to obtain the name, then the character following the one pointed to by the logical length pointer is used as the drive specification, or

   c) If the last character obtained from the string is physically the last character in the string, then the drive number is unspecified.

-- If the character used as the drive specification is not an ASCII digit (0 through 9), then the drive number is unspecified.

-- If the drive number is unspecified, all drives will be searched for the file (starting with drive 0 and ending with the highest numbered drive that is on-line).

-- If the character used as the drive specification is an ASCII digit, then only the drive with that number will be searched to find the file.

-- If the specified drive is off-line an I/O error will occur.

-- Any number of logical files may be open at one time.

-- If the specified logical file is already open, the equivalent of a CLOSE instruction is executed before proceeding with the OPEN.

-- An attempt to OPEN a file that does not exist will result in an I/O error.

-- Executing the OPEN instruction initializes the logical file without changing the disk file in any way.

-- Space compression is turned on by the execution of an OPEN instruction.

Assume that the following statements were included in the program previous to the statements in all of the following examples:

```
FILE     FILE
FILENAME INIT      "PAYROLL11"
```

Example:

```
        RESET     FILENAME TO 9        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 9
.
        RESET     FILENAME TO 4        SET THE FORMPOINTER TO 4
        OPEN      FILE,FILENAME
```

this OPEN instruction will find and initialize a file named ROLL11/TXT on any drive on which it can be found.

Example:

```
        RESET     FILENAME TO 8        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 8
.
        RESET     FILENAME TO 4        SET THE FORMPOINTER TO 4
        OPEN      FILE,FILENAME
```

this OPEN instruction will try to find and initialize a file named ROLL1/TXT from drive 1.

Example:

```
        RESET     FILENAME TO 8        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 8
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        OPEN      FILE,FILENAME
```

this OPEN instruction will try to find and initialize a file named PAYROLL1/TXT from drive 1.

Example:

```
        RESET     FILENAME TO 9        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 9
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        OPEN      FILE,FILENAME
```

this OPEN instruction will try to find and initialize a file named PAYROLL1/TXT from drive 1.

Example:

```
        RESET     FILENAME TO 7        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 7
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        OPEN      FILE,FILENAME
```

this OPEN instruction will try to find and initialize a file named PAYROLL/TXT from drive 1.

Example:

```
        RESET     FILENAME TO 3        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 3
.                   .
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        OPEN      FILE,FILENAME
```

this OPEN instruction will try to find and initialize a file named PAY/TXT from any drive on which it can be found.

12.3.2 CLOSE (General)

This instruction is used to return any unused, newly allocated disk space to DOS for use by another file. CLOSE may have one of the following general formats:

```
1)   <label>  CLOSE    <file>
2)   <label>  CLOSE    <ifile>
3)   <label>  CLOSE    <rfile>
4)   <label>  CLOSE    <rifile>
```

where: <label> is an execution label (see section 2.).
<file> is a file declared using the FILE declaration (see section 5.1).

<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rfile> is a file declared using the RFILE declaration (see section 5.3).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- If only reads or updates were performed on the file, the CLOSE instruction does not need to be used.

-- The equivalent of a CLOSE instruction is automatically performed when one opens a logical file that is already open.

-- Execution of the CLOSE instruction does not write an end-of-file mark to the file.

-- Closing a file from another port could affect the file being used at your port.

-- Execution of the CHAIN instruction (see section 6.8), causes all logical files that are open to be automatically closed without space deallocation being performed. Note that this means files cannot be held open across program chains.

-- A potential problem exists when the CLOSE instruction is performed on files that are in use by more than one port. There is a discussion of this problem in Appendix D.

12.3.3 READ (General)

The READ instruction is used to get information saved on the disk into variables in a Databus program. This instruction may have one of the following general formats:

```
1)   <label>  READ    <file>,<nvar>;<list>
2)   <label>  READ    <ifile>,<nvar>;<list>
3)   <label>  READ    <ifile>,<svar>;<list>
4)   <label>  READ    <rfile>,<nvar>;<list>
5)   <label>  READ    <rifile>,<nvar>;<list>
6)   <label>  READ    <rifile>,<svar>;<list>
```

where: <label> is an execution label (see section 2.).

<nvar> is a numeric variable (see section 4.1).
<svar> is a string variable (see section 4.2).
<file> is a file defined using the FILE declaration (see section 5.1).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rfile> is a file defined using the RFILE declaration (see section 5.3).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be read from the disk.

Programming Considerations:

-- <label> is optional.

-- Formats (1), (2), (4) and (5) are used to read from the disk using one of the following access methods.

   a) If the value of <nvar> < 0, a logical record is read.

   b) If the value of <nvar> > 0 or = 0, a physical record is read.

-- Formats (3) and (6) are used to read indexed records from the disk.

-- The items in the list must be separated by commas.

-- Space decompression is always in effect when doing READ's.

-- If all of the items of the list have been used before the logical end of the record is reached, one of the following actions take place.

   a) If a semicolon is placed at the end of the list, the position within the file is left unchanged after the last item in the list is processed. This allows subsequent I/O operations to pick up at the position where the READ finished.

   b) If a semicolon is not placed at the end of the list, the position within the file is bumped past the next logical end-of-record character (015). This allows subsequent I/O operations to pick up at the start of the next logical record.

-- <list> may be made up of any combination of the following items:

   a) <svar>, a character string va.iable (see section 4.2).

   b) <nvar>, a numeric string variable (see section 4.1).

   c) *<nvar>, a list control (see sections 13.4.1).

   d) *<dnum>, a list control (see sections 12.1.3).

-- If an attempt is made to read a record which has never before been written, the following actions occur.

   a) The position within the file is unchanged.

   b) A RANGE trap occurs.

-- An attempt to read an end-of-file mark (see section 12.1.3) will cause the following actions.

   a) The OVER flag will be set to true.

   b) All numeric string variables in the list will be set to zero.

   c) All character string variables in the list will have:

      1. the formpointer set to zero.

      2. the logical length pointer set to zero.

      3. all of the characters in the variable replaced with blanks.

   d) A semicolon at the end of the READ list has no effect.

   e) The position within the file is reset to point to the end-of-file mark, after processing of the READ is complete. This means that; if the OVER condition flag is ignored, subsequent reads will read the same end-of-file mark.

12.3.3.1 Character String Variables (READ)

When a character string variable appears in the list of a WRITE instruction, characters are read from the disk and put into the variable. These characters are put into the variable as described below.

Programming Considerations:

-- Characters are read from the disk starting at the current position within the file.

-- Characters are stored consecutively starting at the physical beginning of the string variable.

-- Characters are read and stored until the physical end of the character string variable is reached.

-- The formpointer is set to one.

-- The logical length is set to point to the last physical character in the string.

-- If the end of the logical record is encountered while filling a character string variable, the following takes place:

   a) The logical end-of-record character (015) is not stored in the variable.

   b) The logical length pointer of the variable is set to point to the last character stored in the variable.

   c) The suffix of the variable will be filled with blanks.

   These actions are particularly useful when dealing with space compressed files. The trailing blanks deleted by using space compression are restored in this way. (b) above makes it possible to take advantage of the *+ control with DISPLAY and PRINTing of logical records.

-- If the logical end of record is encountered before all of the character string variables in the list are filled, the following actions are taken.

   a) The formpointers of all of the remaining character string variables are set to zero.

   b) The logical length pointers of all of the remaining

character string variables are set to zero.

c) All of the remaining character string variables are filled with blanks.

12.3.3.2 Numeric String Variables (READ)

When a numeric string variable appears in the list of a READ instruction, characters are read from the disk and put into the variable. These characters are put into the variable as described below.

Programming Considerations:

-- Characters are read from the disk starting at the current position within the file.

-- Characters are stored consecutively starting at the physical beginning of the numeric variable.

-- Characters are read and stored until the physical end of the character string variable is reached.

-- Any non-leading spaces read will be converted to zeros (e.g. s3s2s1, where s stands for a space, would be read as s30201).

-- ASCII digits are the only characters accepted with the following exceptions. A FORMAT trap will occur if the following rules are not satisfied.

a) Blanks are always accepted.

b) A minus sign is accepted only when it is the first non-blank character to be read.

c) A minus sign is accepted only when there is room for at least one character to the left of the decimal point.

d) A period is accepted only if the format of the variable calls for a decimal point.

e) Only one period will be accepted.

f) The number of characters that will be accepted before a period is required

equals

the number of places preceding the decimal point in the format of the variable.

g) The number of characters that will be accepted after the period

equals

the number of places following the decimal point in the format of the variable.

h) The last character to be accepted may be a "minus-overpunch" character (see section 12.3.4.3.4). If it is, the character to the left of the most significant digit will contain the sign. If there is already a sign or if there is no room for the sign, 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.

-- If a FORMAT trap occurs during a read, the position within the file is reset to what it was before the READ was attempted.

-- If the end of the logical record is encountered while filling a numeric string variable, the rest of the variable is padded with zeros. Note that if one of these locations within the variable is the decimal point, a FORMAT trap will occur.

-- If the logical end of record is encountered before all of the numeric string variable in the list are filled, all of the remaining variables are set to zero.

12.3.4 WRITE (General)

The WRITE instruction is used to put the information to be saved onto the disk. This instruction may have one of the following general formats:

```
1)  <label>  WRITE   <file>,<nvar>;<list>
2)  <label>  WRITE   <ifile>,<nvar>;<list>
3)  <label>  WRITE   <ifile>,<svar>;<list>
4)  <label>  WRITE   <rfile>,<nvar>;<list>
5)  <label>  WRITE   <rifile>,<nvar>;<list>
6)  <label>  WRITE   <rifile>,<svar>;<list>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).

<svar> is a character string variable (see section 4.2).
<file> is a file defined using the FILE declaration (see section 5.1).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rfile> is a file defined using the RFILE declaration (see section 5.3).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- Formats (1), (2), (4) and (5) are used to write to the disk using one of the following accessing methods.

   a) If the value of <nvar> < 0, a logical record is written.

   b) If the value of <nvar> > 0 or = 0, a physical record is written.

-- Formats (3) and (6) are used to write indexed records to the disk.

-- The items in the list must be separated by commas.

-- <list> may be made up of any combination of the following items:

   a) <svar>, a character string variable (see section 4.2).

   b) <nvar>, a numeric string variable (see section 4.1).

   c) <occ>, an octal control character (see section 2.5).

   d) <list control>, used to control the manner in which the list is processed.

   e) <slit>, a literal of the form "<string>" (see section 2.5). <string> must be a valid character string (see section 4.2).

   f) <nlit>, a literal of the form "<string>" (see section 2.5). <string> must be a valid numeric string (see section 4.1).

12.3.4.1 Character String Variables (WRITE)

When a character string variable appears in the list of a WRITE instruction, the characters saved in the variable are written on the disk. Unless modified by a list control the manner in which the characters are put on the disk is described below.

Programming Considerations:

-- The characters in the variable are written starting with the first physical character and continuing through the logical length.

-- Blanks will be written for any character positions that exist between the logical length pointer and the physical end of the variable.

-- The first character written will be written at the current position within the file.

-- The position within the file is bumped by 1 for each character written. For more details on bumping the position within a file, see section 12.2.

-- The character pointer is left positioned to the right of the last character written.

-- The control characters (formpointer, logical length pointer and 0203) are not written to the disk.

12.3.4.2 Numeric String Variables (WRITE)

When a numeric string variable appears in the list of a WRITE instruction, the characters saved in the variable are written on the disk. Unless modified by a list control the manner in which the characters are put on the disk is described below.

Programming Considerations:

-- The characters in the variable are written starting with the first physical character and continuing through the physical end of the variable.

-- The first character written will be written at the current

position within the file.

-- The position within the file is bumped by 1 for each character written. For more details on bumping the position within a file, see section 12.2.

-- The character pointer is left positioned to the right the last character written.

-- The control characters, 0200 and 0203 (octal), are not written to the disk.

12.3.4.3 List Controls (WRITE)

The list controls are provided to allow more flexibility in the way records are formated. They may be used to control the manner in which variables are written to the disk. All list controls begin with an asterisk followed by the specification of the control function.

12.3.4.3.1 *+ (Space Compression On)

The *+ control may be used to enable space compression. For more details about space compression, see section 12.1.2.

12.3.4.3.2 *- (Space Compression Off)

The *- control may be used to disable space compression. For more details about space compression, see section 12.1.2.

12.3.4.3.3 *ZF (Zero Fill)

This control is used to cause numeric variables to be written with zero fill on the left.

Programming Considerations:

-- This control affects only the first variable following the *ZF in the WRITE list.

-- Zeros are written in place of any leading blanks in the variable.

-- If the variable contains a leading minus sign, the minus sign is written in the leftmost position.

-- The *ZF control when used in conjunction with the *MP control (see section 12.3.4.3.4), causes the minus sign to be replaced with a zero.

12.3.4.3.4 *MP (Minus Overpunch)

The control *MP converts a numeric variable to a "minus over-punch" format.

Programming Considerations:

-- This control affects only the first variable following the *MP.

-- This control affects only numeric variables that have a negative value.

-- The minus sign is over punched over the right-most digit.

-- The right-most digit written to the disk is as follows:

   a) If the right-most digit is a zero, it is converted to a right bracket "}".

   b) One through nine convert to "J" through "R". "1" becomes "J", "2" becomes "K", "3" becomes "L", and so on.

12.3.4.4 Octal Control Characters

Octal control characters are written to the disk exactly as they appear in the WRITE list.

Programming Considerations:

-- The control character will be written at the current position within the file.

-- The position within the file is bumped by 1. For more details on bumping the position within a file, see section 12.2.

-- Caution should be exercised when using octal control

characters. Some of the control characters (000, 003, 011 and 015) have special meaning to the READ instruction and their use can cause confusion.

12.3.4.5 Literals

When a literal (<slit> or <nlit>) appears in the list of a WRITE instruction, that literal is written to the disk.

Programming Considerations:

-- All of the characters between the double quotes are written as they appear in the literal.

-- The first character written will be written at the current position within the file.

-- The position within the file is bumped by 1 for each character written. For more details on bumping the position within a file, see section 12.2.

-- The character pointer is left positioned to the right of the last character written.

CHAPTER 13. PHYSICAL RECORD ACCESSING

The following sections discuss the aspects of the Disk I/O instructions that apply to accessing physical records only.

13.1 OPEN (Physical)

The following sections discuss the aspects of the OPEN instruction that apply to accessing physical records only. For a general discussion of the OPEN instruction, see section 12.3.1. One of the following general formats may be used:

```
1)   <label>  OPEN     <file>,<slit>
2)   <label>  OPEN     <file>,<svar>
3)   <label>  OPEN     <rfile>,<slit>
4)   <label>  OPEN     <rfile>,<svar>
```

where: <label> is an execution label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<svar> is a string variable (see section 4.2).
<file> is a file declared using the FILE declaration (see section 5.1).
<rfile> is a file declared using the RFILE declaration (see section 5.3).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- See section 12.3.1.

-- The position within the file is initialized to:

   a. Record number = 0.

   b. Character pointer = 1.

13.2 PREPARE (PREP) (Physical)

This instruction is used to create and initialize a logical file for use by a Databus program. One of the following general formats may be used:

```
1)   <label>  PREPARE   <file>,<slit>
2)   <label>  PREPARE   <file>,<svar>
3)   <label>  PREPARE   <rfile>,<slit>
4)   <label>  PREPARE   <rfile>,<svar>
```

where: <label> is an execution label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<svar> is a string variable (see section 4.2).
<file> is a file declared using the FILE declaration (see section 5.1).
<rfile> is a file declared using the RFILE declaration (see section 5.3).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- The value of <svar> is unchanged by the execution of this instruction.

-- The string literal, when using format (1) or (3); specifies the DOS name of the disk file to be associated with the label.

-- The string variable, when using format (2) or (8); specifies the DOS name of the disk file to be associated with the label.

-- PREPARE is identical to the OPEN instruction (see section 13.1) with the following exceptions:

   a. PREPARE cannot be used with indexed files.

   b. If the file cannot be found, then instead of giving an I/O error, a new file is created.

   c. If a new file is to be created, it is put on the disk drive decribed below.

      1. If the drive number is specified in the string variable or literal, it is put on that drive.

2. If the drive number is unspecified, it is put on drive 0.

d. If the file to be prepared already exists and is delete or write protected, an I/O error will occur.

-- If the user plans to deal with a very large file, he should write a dummy record into the largest record number he plans to use. This allows DOS to allocate all of the sectors for that file in the most optimal manner possible. Physical record accessing becomes that much faster.

Assume that the following statements were included in the program previous to the statements in all of the following examples:

```
FILE     FILE
FILENAME INIT      "PAYROLL11"
```

Also, assume that the specified files need to be created and do not already exist.

Example:

```
RESET    FILENAME TO 9       SET THE LOGICAL
LENSET   FILENAME            LENGTH POINTER TO 9

RESET    FILENAME TO 4       SET THE FORMPOINTER TO 4
PREP     FILE,FILENAME
```

this PREP instruction will create a file named ROLL11/TXT on drive 0.

Example:

```
RESET    FILENAME TO 8       SET THE LOGICAL
LENSET   FILENAME            LENGTH POINTER TO 8

RESET    FILENAME TO 4       SET THE FORMPOINTER TO 4
PREP     FILE,FILENAME
```

this PREP instruction will create a file named ROLL1/TXT on drive 1.

Example:

```
        RESET     FILENAME TO 8        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 8
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        PREP      FILE,FILENAME
```

this PREP instruction will create a file named PAYROLL1/TXT on drive 1.

Example:

```
        RESET     FILENAME TO 9        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 9
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        PREP      FILE,FILENAME
```

this PREP instruction will create a file named PAYROLL1/TXT on drive 1.

Example:

```
        RESET     FILENAME TO 7        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 7
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        PREP      FILE,FILENAME
```

this PREP instruction will create a file named PAYROLL/TXT on drive 1.

Example:

```
        RESET     FILENAME TO 3        SET THE LOGICAL
        LENSET    FILENAME             LENGTH POINTER TO 3
.
        RESET     FILENAME TO 1        SET THE FORMPOINTER TO 1
        PREP      FILE,FILENAME
```

this PREP instruction will create a file named PAY/TXT on drive 0.

13.3 CLOSE (Physical)

This instruction is used to return any unused, newly allocated disk space to DOS for use by another file. CLOSE is also used along with PREPARE to delete a file from the disk file structure. The following sections discuss the aspects of the CLOSE instruction that apply to accessing physical records only. For a general discussion of the CLOSE instruction, see section 12.3.2. CLOSE may have one of the following general formats:

```
1)   <label>  CLOSE    <file>
2)   <label>  CLOSE    <rfile>
```

where: <label> is an execution label (see section 2.).
<file> is a file declared using the FILE declaration (see section 5.1).
<rfile> is a file declared using the RFILE declaration (see section 5.3).

Programming Considerations:

-- <label> is optional.

-- See section 12.3.2.

-- CLOSE when used in conjunction with the PREPARE instruction (see section 13.2) is used to delete a file from the DOS file system. If the PREPARE instruction is immediately followed by a CLOSE instruction, the file described in the PREPARE instruction will be deleted from the DOS file system.

13.4 READ (Physical)

The READ instruction is used to get information saved on the disk into variables in a Databus program. The following sections discuss the aspects of the READ instruction that apply to accessing physical records only. For a general discussion of the READ instruction, see section 12.3.3. This instruction may have one of the following general formats:

```
1)   <label>  READ     <file>,<nvar>;<list>
2)   <label>  READ     <rfile>,<nvar>;<list>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).

<rfile> is a file defined using the RFILE declaration (see section 5.3).
<list> is a list of items describing the information to be read from the disk (see section 12.3.3).

Programming Considerations:

-- <label> is optional.

-- See section 12.3.3.

-- The first action taken by the READ instruction, is to reset the position within the file as follows:

   a) The record number is set to the value given in <nvar>. (All digits after the decimal point are ignored.)

   b) The character pointer is set to 1.

-- Since reading a physical record always resets the position within the file before the READ continues, it is unnecessary to continue scanning until the next logical record is reached. This extra scanning for the 015 (end-of-record) is not only unnecessary but uses extra processor time. It is a simple matter of putting a semi-colon at the end of the read list to eliminate this wasted processing.

Example:

```
FDECL   FILE
RN      FORM    " 2.6"
        OPEN    FDECL,"DATA"
        READ    FDECL,RN;A,B,C
```

This READ instruction could be used to read from file DATA/TXT the values of variables A, B and C. The position within file DATA/TXT would first be established at record number 2 with a character pointer of 1. Variables A, B and C are then read. Any remaining characters in the logical record will be ignored and the position within the file will be left at the beginning of the next logical record.

Example:

```
FDECL   FILE
RN      FORM    " 2.6"
        OPEN    FDECL,"DATA"
        READ    FDECL,RN;A,B,C;
```

This READ instruction is similar to the one in the above example except that the position within the file will be left at the character after the last one read into the variable C.

Example:

```
FDECL    FILE
REWIND   FORM      " 0"
         OPEN      FDECL,"DATA"
         READ      FDECL,REWIND;;
```

This READ instruction will establish the same position within the file exactly as if an OPEN or PREP instruction had just been executed. The first action is to set the position within the file to record 0 with the character pointer equal to 1. Because of the second semi-colon as the list terminator, the position will not be bumped to the next logical record on termination of the execution of the READ.

-- <list> may be made up of any combination of the following items:

   a) <svar>, a character string variable (see section 12.3.3.1).

   b) <nvar>, a numeric string variable (see section 12.3.3.2).

   c) <tab control>, a list control which is used to tab to the position within the record where the data is to be obtained.

13.4.1 Tab Control

Tabbing is a feature which can eliminate unwanted data transfers to and from the disk controller buffer. It also allows the programmer to save considerable space in his data area. The tab control may have one of the following general formats:

1) *<nvar>
2) *<dnum>

where: <nvar> is a numeric variable (see section 4.1).
<dnum> is a decimal number.

-- When format (1) is used, the value of the numeric variable specifies the tab position.

-- When format (2) is used, the decimal number specifies the tab position.

-- The character pointer is set to the specified tab position.

-- Tabbing can be used only when the logical records do not cross physical record boundaries. This condition can usually be enforced through the use of the DOS REFORMAT utility and careful use of Databus WRITE instructions.

-- An attempt to tab past the physical end-of-record will result in an I/O error.

-- Using tabbing may cause the READ instruction to fail to recognize an EOF mark. The EOF mark can be recognized only when READ is positioned to character position 1, followed by an attempt to read a variable.

-- Tab positioning on physical accesses is always calculated from the first character position in the current physical record.

Example:

```
FDECL    FILE
RN       FORM      " 3.3"
TAB      FORM      "25"
         OPEN      FDECL,"DATA"
         READ      FDECL,RN;A,*100,B,*TAB,C,*50,D;
```

The READ instruction in this example would set the record number to 3 and the character pointer to 1. Variable A would then be read. Next the character pointer would be set to 100 and variable B would be read. The character pointer would then be set to 25 and variable C would be read. Finally, the character pointer would be set to 50 and variable D would be read. The character pointer would be left pointing after the last character read into variable D since the semicolon appears at the end of the list.

13.5 WRITE (Physical)

The WRITE instruction is used to put the information to be saved onto the disk. The following sections discuss the aspects of the WRITE instruction that apply to accessing physical records only. For a general discussion of the WRITE instruction, see section 12.3.4. This instruction may have one of the following general formats:

```
1)   <label>  WRITE     <file>,<nvar>;<list>
2)   <label>  WRITE     <file>,<nvar>;<list>;
3)   <label>  WRITE     <rfile>,<nvar>;<list>
4)   <label>  WRITE     <rfile>,<nvar>;<list>;
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- See section 12.3.4.

-- The first action taken by the WRITE instruction, is to reset the position within the file as follows:

   a) The record number is set to the value given in <nvar>. (All digits after the decimal point are ignored.)

   b) The character pointer is set to 1.

-- Processing for the WRITE instruction is terminated as follows:

   a) Formats (1) and (3) cause:

      1) an 015 (logical end of record character) to be written,

      2) the position within the file to be bumped by 1, and

      3) an 003 (physical end of record character) to be written.

4) The character pointer is left pointing to the 003 character.

b) Formats (2) and (4) cause the position within the file to be unchanged after processing the last item in the list. This operation is useful for writing the first part of a record where more of the record will be written later.

-- Tab positioning is not allowed when using WRITE instructions. If tabbing is required while writing to the disk, the WRITAB instruction should be used.

13.6 WRITAB

The WRITAB instruction allows tabbing while modifying a physical record. WRITAB allows characters to be written into any character position of a physical record without disturbing the rest of the record. This instruction may have one of the following general formats:

```
1)   <label>  WRITAB    <file>,<nvar>;<list>
2)   <label>  WRITAB    <rfile>,<nvar>;<list>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- Executing a WRITAB instruction is equivalent to executing one of the following WRITE instruction except that tabbing is allowed.

```
<label>  WRITE    <file>,<nvar>;<list>;
<label>  WRITE    <rfile>,<nvar>;<list>;
```

A separate mnemonic is required for tabbed writes because it is necessary to do an additional disk read when tabbing is to be used.

-- If an attempt is made to read a record which has never before been written, the following actions occur.

   a) The position within the file is unchanged.

   b) A RANGE trap occurs.

-- WRITAB allows tab controls to be used as items in the list.

13.6.1 Tab Control

Tabbing is a feature which can eliminate unwanted data transfers to and from the disk controller buffer. It also allows the programmer to save considerable space in his data area. The tab control may have one of the following general formats:

1) *<nvar>
2) *<dnum>

where: <nvar> is a numeric variable (see section 4.1).
<dnum> is a decimal number.

-- When format (1) is used, the value of the numeric variable specifies the tab position.

-- When format (2) is used, the decimal number specifies the tab position.

-- The character pointer is set to the specified tab position.

-- Tabbing can be used only when the logical records do not cross physical record boundaries. This condition can usually be enforced through the use of the DOS REFORMAT utility and careful use of Databus WRITE instructions.

-- An attempt to tab past the physical end-of-record will result in an I/O error. Caution: While tabbing beyond the end of record is not allowed, any other list item could cause the logical record to extend across a physical record boundary.

-- Tab positioning on physical accesses is always calculated from the first character position in the current physical record.

-- If the record number is bumped while processing a list item other than a tab control, subsequent tabs will position into the new physical record, not the original one.

Example:

```
FDECL    FILE
RN       FORM      " 3.3"
TAB      FORM      "25"
         OPEN      FDECL,"DATA"
WRITAB   FDECL,RN;A,*100,B,*TAB,C,*50,D;
```

The WRITAB instruction in this example would set the record number to 3 and the character pointer to 1. Variable A would then be written over those characters already in the record. Next the character pointer would be set to 100 and variable B would be written. The character pointer would then be set to 25 and variable C would be written. Finally, the character pointer would be set to 50 and variable D would be written. The character pointer would be left pointing after the last character written from variable D since there is always an implied semicolon at the end of the list. The characters already in the disk record at those positions that were not overwritten will remain unchanged.

13.7 WEOF

The WEOF instruction allows a DOS end of file mark (see section 12.1.3) to be written to a file. This instruction has the following general format:

```
1)   <label>  WEOF      <file>,<nvar>
2)   <label>  WEOF      <rfile>,<nvar>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).

Programming Considerations:

-- <label> is optional.

-- The EOF mark is written to the record number specified in the numeric variable. (All digits after the decimal point are ignored.)

-- The position within the file is left at the beginning of the

EOF that was written.

CHAPTER 14. LOGICAL RECORD ACCESSING

The following sections discuss the aspects of the Disk I/O instructions that apply to accessing logical records only.

14.1 OPEN (Logical)

All of the aspects of opening a file for use with logical record accessing are identical to those used with physical record accessing (see Section 13.1).

14.2 PREPARE (Logical)

All of the aspects of preparing a file for use with logical record accessing are identical to those used with physical record accessing (see Chapter 13.2).

14.3 CLOSE (Logical)

All of the aspects of closing a file for use with logical record accessing are identical to those used with physical record accessing (see Chapter 13.3).

14.4 READ (Logical)

The READ instruction is used to get information saved on the disk into variables in a Databus program. The following sections discuss the aspects of the READ instruction that apply to accessing logical records only. For a general discussion of the READ instruction, see section 12.3.3. This instruction may have one of the following general formats:

```
1)   <label>  READ      <file>,<nvar>;<list>
2)   <label>  READ      <rfile>,<nvar>;<list>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).

<list> is a list of items describing the information to be read from the disk (see section 12.3.3).

Programming Considerations:

-- <label> is optional.

-- <nvar> must have a negative value.

-- See section 12.3.3.

-- Reading starts at the current position within the file. That is, the READ starts where any previous disk I/O operation on the file left the position.

-- <list> may be made up of any combination of the following items:

   a) <svar>, a character string variable (see section 12.3.3.1).

   b) <nvar>, a numeric string variable (see section 12.3.3.2).

   c) <tab control>, a list control which is used to tab to the position within the record where the data is to be obtained.

-- Using the tab controls when reading logical records is possible but not advisable. Since the tab position is calculated relative to the start of the physical record and not the start of the logical record, using a tab control could tab into a different logical record.

Example:

```
FDECL    FILE
SEQ      FORM      "-1"
         OPEN      FDECL,"DATA"
         READ      FDECL,SEQ;A,B,C
```

Variables A, B, and C would be read starting at the current position within the file. Any remaining characters in the logical record will be ignored and the position within the file will be left at the beginning of the next logical record.

Example: This program will list DATA/TXT on the screen.

```
FDECL    FILE
SEQ      FORM      "-1"
```

```
LINE     DIM       80
.
         OPEN      FDECL,"DATA"
.
LOOP     READ      FDECL,SEQ;LINE
         STOP      IF OVER
         DISPLAY   *R,*P1:24,*+,LINE
         GOTO      LOOP
```

14.5 WRITE (Logical)

The WRITE instruction is used to put the information to be saved onto the disk. The following sections discuss the aspects of the WRITE instruction that apply to accessing logical records only. For a general discussion of the WRITE instruction, see section 12.3.4. This instruction may have one of the following general formats:

```
1)   <label>  WRITE     <file>,<nvar>;<list>
2)   <label>  WRITE     <file>,<nvar>;<list>;
3)   <label>  WRITE     <rfile>,<nvar>;<list>
4)   <label>  WRITE     <rfile>,<nvar>;<list>;
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- <nvar> must have a negative value.

-- See section 12.3.4.

-- Characters are put on the disk starting at the current position within the file being referenced. That is, the READ starts where any previous disk I/O operation on the file left the position.

-- Processing for the WRITE instruction is terminated as follows:

a) Formats (1) and (3) cause:

   1) an 015 (logical end of record character) to be written, and

   2) the position within the file to be bumped by 1.

b) Formats (2) and (4) cause the position within the file to be unchanged after processing the last item in the list. This operation is used only for writing the first part of a record where more of the record will be written later.

-- Tab positioning is not allowed when using WRITE instructions. If tabbing is required while writing to the disk, the WRITAB instruction should be used.

14.6 WRITAB (Logical)

Using tab positioning when writing logical records is possible but not advisable. Since the tab position is calculated relative to the start of the physical record and not the start of the logical record, using a tab control could tab into a different logical record.

The only difference between using WRITAB on logical records rather than physical records is that the current record number is used to determine which physical record will be modified.

14.7 WEOF (Logical)

The WEOF instruction allows a DOS end of file mark (see section 12.1.3) to be written to a file. This instruction has the following general format:

```
1)   <label>  WEOF      <file>,<nvar>
2)   <label>  WEOF      <rfile>,<nvar>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<file> is a file defined using the FILE declaration (see section 5.1).
<rfile> is a file defined using the RFILE declaration (see section 5.3).

Programming Considerations:

-- <label> is optional.

-- <nvar> must have a negative value.

-- If the current position within the file is at the beginning of a physical record, the EOF is written into that record.

-- If the current position within the file is not at the beginning of a physical record, the following actions are taken:

   a) A physical end of record character (003) is written at the current position, and

   b) The position within the file is bumped to the next physical record and the EOF is written into that record.

-- The position within the file is left at the beginning of the EOF that was written.

CHAPTER 15. INDEXED RECORD ACCESSING

The following sections discuss the aspects of the Disk I/O instructions that apply to accessing indexed records only.

15.1 OPEN (Indexed)

The following sections discuss the aspects of the OPEN instruction that apply to accessing indexed records only. For a general discussion of the OPEN instruction, see section 12.3.1. One of the following general formats may be used:

```
1)   <label>  OPEN     <ifile>,<slit>
2)   <label>  OPEN     <ifile>,<svar>
3)   <label>  OPEN     <rifile>,<slit>
4)   <label>  OPEN     <rifile>,<svar>
```

where: <label> is an execution label (see section 2.).
<slit> is a literal of the form "<string>" (see section 2.5).
<svar> is a string variable (see section 4.2).
<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- <slit> must be a valid character string (see section 4.2).

-- See section 12.3.1.

-- OPEN initializes both the index file and the data file that has been indexed.

-- If the drive number is specifed (see section 12.3.1), both the index file and the data file must be on the specified drive.

-- If the drive number is not specified (see section 12.3.1), the index file and the data file may be on different drives.

-- The name of the data file to be opened is contained in the

index file.

-- Opening the index file automatically causes the data file to be opened.

-- If the data file is indexed by more than one index file, then each index file must be opened using a different logical file.

-- The position within the data file is initialized to:

   a. Record number = 0.

   b. Character pointer = 1.

-- The position within the index file is left undefined.

Assume that the following statements were included in the program previous to the statements in all of the following examples:

```
        DECL    IFILE
```

Also, assume that index files, DATA/ISI and DATA2/ISI, have been created by indexing the data file, DATA/TXT, using the DOS INDEX utility as shown below:

```
INDEX DATA/TXT:DR0,DATA/ISI:DR0;1-5
INDEX DATA/TXT:DR0,DATA2/ISI:DR1;6-10
```

Note that DATA/TXT is on drive 0, DATA/ISI is on drive 0 and DATA2/ISI is on drive 1.

Example:

```
        OPEN    DECL,"DATA    0"
```

This OPEN instruction will initialize DATA/ISI and DATA/TXT on drive 0.

Example:

```
        OPEN    DECL,"DATA    1"
```

This OPEN instruction will cause an I/O error, since neither DATA/ISI nor DATA/TXT are on drive 1.

Example:

```
OPEN     DECL,"DATA"
```

This OPEN instruction will initialize DATA/ISI and DATA/TXT on drive 0.

Example:

```
OPEN     DECL,"DATA2   0"
```

This OPEN instruction will cause an I/O error, since DATA2/ISI is not on drive 0.

Example:

```
OPEN     DECL,"DATA2   1"
```

This OPEN instruction will cause an I/O error, since DATA/TXT is not on drive 1.

Example:

```
OPEN     DECL,"DATA2"
```

This OPEN instruction will initialize DATA2/ISI on drive 1 and DATA/TXT on drive 0.

15.2 CLOSE (Indexed)

This instruction is used to return any unused, newly allocated disk space to DOS for use by another file. The following sections discuss the aspects of the CLOSE instruction that apply to accessing indexed records only. For a general discussion of the CLOSE instruction, see section 12.3.2. CLOSE may have one of the following general formats:

1) <label> CLOSE <ifile>
2) <label> CLOSE <rifile>

where: <label> is an execution label (see section 2.).
<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- See section 12.3.2.

-- Only the data file is affected by executing the CLOSE instruction.

-- The index file is unchanged by the execution of the CLOSE instruction.

15.3 READ (Indexed)

The READ instruction is used to get information saved on the disk into variables in a Databus program. The following sections discuss the aspects of the READ instruction that apply to accessing indexed records only. For a general discussion of the READ instruction, see section 12.3.3. This instruction may have one of the following general formats:

```
1)   <label>  READ     <ifile>,<nvar>;<list>
2)   <label>  READ     <ifile>,<svar>;<list>
3)   <label>  READ     <rifile>,<nvar>;<list>
4)   <label>  READ     <rifile>,<svar>;<list>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<svar> is a string variable (see section 4.2).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be read from the disk.

Programming Considerations:

-- <label> is optional.

-- The following apply when formats (1) and (3) are used:

a) The READ instruction accesses only the data file.

b) The READ is either a physical access (see section 13.4) or a logical access (see section 14.4).

c) The index file is not used or modified in any way by the

READ.

-- The logical string of <svar> specifies the key to be used when searching the index file.

-- The key is considered to match an item in the index file (an index item is a key in the index file) if one of the following rules hold true.

   a) If both the key and the index item have the same number of characters, all of the characters must match.

   b) If the key has more characters than the index item, then:

      1) All of the characters up through the length of the index item must match, and

      2) the remaining characters of the key must be blanks.

   c) If the key has less characters than the index item, there is no match.

-- If a match is found,

   a) The position of the logical record to be accessed is obtained from the index file. The position within the data file is then initialized to this value.

   b) Once the position within the data file is established, the READ proceeds precisely as if it were a logical record access (see section 14.4). (Exception: see the Programming Consideration below concerning tab positioning.)

   c) The position within the index file is initialized to the next item in sequence in the index file.

-- If no match is found,

   a) the OVER condition flag is set to true,

   b) all of the variables in the list are unchanged, and

   c) the position within the index file is left pointing to the next item in sequence in the index file.

-- The OVER condition being set after an indexed READ operation indicates that the key specified could not be found in the

index.

-- The test for the OVER condition should be made after the READ statement.

-- Tab positions when using indexed access are calculated relative to the beginning of the logical record instead of relative to the beginning of the physical record.

-- If the key is null, the last indexed record that was accessed is re-read without using the index file to access the record. This saves the time needed to search the index file for the key. When the same indexed record needs to be read several times, this feature may save considerable time.

-- Using a null key causes an I/O error if there was not a previous successful READ performed using a non-null key.

15.4 WRITE (Indexed)

The WRITE instruction is used to put the information to be saved onto the disk. The following sections discuss the aspects of the WRITE instruction that apply to accessing indexed records only. For a general discussion of the WRITE instruction, see section 12.3.4. This instruction may have one of the following general formats:

```
1)   <label>  WRITE    <ifile>,<nvar>;<list>
2)   <label>  WRITE    <ifile>,<nvar>;<list>;
3)   <label>  WRITE    <ifile>,<svar>;<list>
4)   <label>  WRITE    <ifile>,<svar>;<list>;
5)   <label>  WRITE    <rifile>,<nvar>;<list>
6)   <label>  WRITE    <rifile>,<nvar>;<list>;
7)   <label>  WRITE    <rifile>,<svar>;<list>
8)   <label>  WRITE    <rifile>,<svar>;<list>;
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<svar> is a character string variable (see section 4.2).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- See section 12.3.4.

-- The following apply when formats (1), (2), (5) and (6) are used:

   a) The WRITE instruction accesses only the data file.

   b) The WRITE is either a physical access (see section 13.5) or a logical access (see section 14.5).

   c) The index file is not used or modified in any way by the WRITE.

-- The logical string of <svar>, when using formats (3), (4), (7) and (8), specifies the key to be inserted into the index file.

-- If the key is null, an I/O error will result.

-- If the key already exists in the index file, an I/O error will result.

-- The search algorithm, used to determine whether the key is already in the index, is identical to that used in the indexed access READ operation (see section 15.3).

-- WRITE uses the following procedure:

   a) The key is inserted into the index such that the keys in the index file remain in collating sequence (alphabetical order).

   b) The data file is searched for its end-of-file mark.

   c) The record is written over the end-of-file mark and proceeds exactly as if it were a physical record write (see section 14.5).

   d) A new end-of-file mark is written to the next physical record.

   e) This implies that for each record inserted into the data file, at least one physical record will be used, no matter how large or small the record.

-- Processing for the WRITE instruction is terminated as follows:

a) Formats (1), (3), (5) and (7) cause:

1) all of the actions taken when terminating a logical record WRITE (see section 14.5), plus

2) the position within the data file to be bumped to the next physical record, and

3) an end-of-file mark to be written.

b) Formats (2), (4), (6) and (8) cause:

1) the position within the file to be unchanged after processing the last item in the list. This operation is useful for writing the first part of a record where more of the record will be written later.

2) The end-of-file mark is not written. This makes it the programmers responsibility to write the end-of-file mark himself.

3) If the programmer fails to write an end-of-file mark, then the next attempt to insert a record will cause a RANGE trap. This insertion will fail because the search for the end-of-file mark will fail.

-- Timing considerations:

a) Inserting many records causes indexed accesses to become less random and more sequential. (Random accessing takes much less time than sequential accessing.)

b) Inserting many records whose keys are close together in the collating sequence causes indexed accesses to become less random. (For example: AAAB is much closer to AAAA than BBBB.)

c) Indexed accesses start taking significantly longer when one tenth of the records in an indexed file have been inserted.

d) Generally, use the DOS INDEX utility as often as possible to insure that indexed accesses are as random as possible.

15.5 WEOF (Indexed)

The WEOF instruction allows a DOS end of file mark (see section 12.1.3) to be written to a file. This instruction has the following general format:

```
1)   <label>  WEOF      <ifile>,<nvar>
2)   <label>  WEOF      <rifile>,<nvar>
```

where: <label> is an execution label (see section 2.).
<nvar> is a numeric variable (see section 4.1).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- The WEOF instruction accesses only the data file.

-- The WRITE is either a physical access (see section 13.7) or a logical access (see section 14.7).

-- The index file is not used or modified in any way by the WRITE.

15.6 READKS

The READKS (READ Key Sequential) instruction is provided to allow indexed records to be read in collating sequence (alphabetical order). This instruction has the following general format:

```
1)   <label>  READKS    <ifile>;<list>
2)   <label>  READKS    <rifile>;<list>
```

where: <label> is an execution label (see section 2.).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be read from the disk.

Programming Considerations:

-- <label> is optional.

-- The current position within the index file is used to get a position in the data file.

-- After the position within the data file has been determined from the index file, the position within the index file is bumped to the next key in the collating sequence (alphabetical order). The ASCII collating sequence is used.

-- If the position within in the index file is past the last key in the index:

   a) The OVER condition flag is set to true, and

   b) All of the variables in the list will have an indeterminant value.

-- Except that the initial position within the data file is determined as described above, READKS proceeds identically to an indexed access READ (see section 15.3).

Example:

```
DECL     IFILE                          INDEX FILE DECLARATION
KEY      INIT      "          "         KEY VALUE USED TO
.                                       INITIALIZE THE INDEX FILE
LINE     DIM       80                   LINE BUFFER
         TRAP      NOFILE IF IO         CATCH FILES NOT ON DISK
         OPEN      DECL,"DATA"          LOOK FOR DATA/TXT AND
.                                       DATA/ISI
         TRAPCLR   IO                   OPEN SUCCEEDED SO DON'T
.                                       CATCH ANY MORE ERRORS
         READ      DECL,KEY;;           INITIALIZE THE POSITION
.                                       WITHIN THE INDEX FILE
*
LOOP     READKS    DECL;LINE            READ IN THE LINE POINTED
.                                       TO BY THE NEXT KEY
         STOP      IF OVER              OVER MEANS NO MORE KEYS
         DISPLAY   *R,*P1:12,*+,LINE    DISPLAY THE LINE
         GOTO      LOOP                 GO GET THE NEXT LINE
*
.                  TELL THE OPERATOR SOMETHING IS WRONG
.
NOFILE   DISPLAY   *R,*P1:12,"NO SUCH FILE"
         STOP
```

15.7 UPDATE

The UPDATE instruction allows tabbing while modifying an indexed record. UPDATE allows characters to be written into any character position of an indexed record without disturbing the rest of the record. This instruction may have one of the following general formats:

1) <label> UPDATE <ifile>;<list>
2) <label> UPDATE <rifile>;<list>

where: <label> is an execution label (see section 2.).
<ifile> is a file defined using the IFILE declaration (see section 5.2).
<rifile> is a file defined using the RIFILE declaration (see section 5.4).
<list> is a list of items describing the information to be written to the disk.

Programming Considerations:

-- <label> is optional.

-- UPDATE is used to modify the last indexed record accessed by any indexed record instruction (typically a READ or READKS).

-- With the following exceptions UPDATE functions the same as WRITAB.

a) All tab positions are calculated relative to the beginning of the logical record, rather than relative to the beginning of the physical record.

b) The initial position within the data file is determined as described above, rather than being furnished by a variable.

c) Since UPDATE modifies logical records instead of physical records, it is possible to tab across physical record boundaries.

-- Attempting an UPDATE when no other index operation has been performed prior to the execution of the UPDATE, will cause and I/O error.

-- It is possible to overstore the 015 (logical end of record) and the 003 (physical end of record) characters when using UPDATE. If extreme care is not exercised, this can result in

more than one record being turned into a single very large record. In some cases it can result in an I/O error.

15.8 INSERT

INSERT provides the capability for inserting a single key into more than one index file. This instruction must be used in conjunction with indexed WRITE's. The indexed record is written to the data file by the WRITE instruction, which also inserts the key into one index file. Since the indexed record does not need to be re-written, the INSERT instruction is used to insert the key into any addition index files. One of the following general formats may be used:

```
1)   <label>  INSERT    <ifile>,<svar>
2)   <label>  INSERT    <rifile>,<svar>
```

where: <label> is an execution label (see section 2.).
<svar> is a string variable (see section 4.2).
<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- The logical string of <svar> specifies the key to be inserted.

-- One INSERT must be executed for each index file which will contain the key.

-- If the key is null, an I/O error will result.

-- If the key already exists in the index file, an I/O error will result.

-- The search algorithm, used to determine whether the key is already in the index, is identical to that used in the indexed access READ operation (see section 14.3).

-- The key is inserted into the index such that the keys in the index file remain in collating sequence (alphabetical order).

-- The logical record written to the data file by the most

recently executed indexed access WRITE, is the record which is indexed by the execution of INSERT. Executing another indexed access WRITE destroys the pointer to the indexed record of the previous WRITE.

-- ** WARNING ** executing an INSERT before any indexed WRITE's are executed may result in damage to the data file.

-- It is not necessary to prevent the program from being interrupted between the WRITE and INSERT instructions.

-- Timing considerations:

a) Inserting many records causes indexed accesses to become less random and more sequential. (Random accessing takes much less time than sequential accessing.)

b) Inserting many records whose keys are close together in the collating sequence causes indexed accesses to become less random. (For example: AAAB is much closer to AAAA than BBBB.)

c) Indexed accesses start taking significantly longer when one tenth of the records in an indexed file have been inserted.

d) Generally, use the DOS INDEX utility as often as possible to insure that indexed accesses are as random as possible.

15.9 DELETE

This operation allows a record to be physically deleted from a data file and for its key to be deleted from the specified index. One of the following general formats may be used:

1) <label> DELETE <ifile>,<svar>
2) <label> DELETE <rifile>,<svar>

where: <label> is an execution label (see section 2.).
<svar> is a string variable (see section 4.2).
<ifile> is a file declared using the IFILE declaration (see section 5.2).
<rifile> is a file declared using the RIFILE declaration (see section 5.4).

Programming Considerations:

-- <label> is optional.

-- The logical string of <svar> specifies the key to be deleted.

-- One DELETE must be executed for each index file which will need the key deleted.

-- If the key is null, an I/O error will result.

-- If the key cannot be found in the index, the OVER condition flag is set.

-- The indexed record is deleted by overstoring every character in the record with an 032 (octal). This includes the logical end of record character (015).

-- Both the DOS REFORMAT utility and the Databus interpreters ignore all 032 characters while reading. Therefore, while reading these characters do not appear to exist.

-- The DOS REFORMAT utility may be used to eliminate the 032 control characters from the data file.

-- If the indexed record to be deleted has already been deleted, then the only action taken is to delete the key from the index file.

CHAPTER 16. PROGRAM GENERATION

16.1 Preparing Source Files

Files containing the source language for DOS DATABUS COMPILER programs are prepared using the general purpose editor running under the DOS (the editor's use is covered in the DOS User's Guide). The editor tab stops may be set to be suitable for keyin of DATABUS programs by using the :T command and setting two tabs, one at 10 and the other at 20.

16.2 Compiling Source Files

DATABUS programs are compiled using the DOS DATABUS COMPILER running under the DOS. The DOS DATABUS compiler is parameterized in the following manner:

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

Where:

<source> is the DOS file specification containing the DATABUS source code.

-- If no file extension is specified "/TXT" is assumed.

-- If no drive is specified all drives starting with drive zero (0) are searched for the source file.

<object> is the DOS file specification for the object file.

-- If no file specification is given, then the DATABUS object file name is the same as the source file with extension "/DBC".

-- If a drive number is not specified and the object file does not exist, then the object file is placed on the same drive as the source file. If the object file already exists, the object code is placed on the same drive as the already existing object file.

<print> is the DOS file specification for the print file.

-- If no name is given for the print file specification, the source file name will be assumed. File extension of "/PRT" will be used if none is specified. Note that if the print file is to be read under to interpreter it must have a file extension of "/TXT" since all files accessed by the interpreter must have that extension.

-- If no drive is specified the print file will be placed on the same drive as the source file.

-- A print file may be specified by simply keying a comma (,) after the object file specification or if no object file is specified, by keying in two commas (,,) after the source file specification. Note that the file extension assumed in this case is "/PRT".

-- The print file specification causes any printout requested to be written into this file instead of being printed on the line 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 standart COBOL and FORTRAN print file format).

16.2.1 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 DATABUS program text created with the editor. This file must always be specified. If no

extension is given on the source file name, the extension TXT is assumed. If the source file name is not supplied, the message:

NAME REQUIRED

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

NO SUCH NAME

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

16.2.2 Output Parameters

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

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

Any source code lines which have errors are displayed on the screen during pass 2, with the appropriate error flag. Additionally, the compiler displays at the lower left corner of the screen the current line number being compiled, for every 10th line. Every 10th line is indicated because displaying the line number for every line would slow down the compiler. No numbers will be displayed if the program is fewer than 10 lines long. This line number display is cleared when processing of included files begins or ends, so the line number display will blink off momentarily during compilation of source files using includled files.

To specify output options, a semicolon (;) plus one of more of the following should be placed after the last file specifications:

L A listing of the compilation results is printed. Each line of source code is numbered and the object code

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

C A listing of the compilation results is printed and the generated object code is listed to the left of the source code. Printing the object code usually makes the listing about twice as long. If this option is given, the L option is implied and need not also be given.

E The source code for lines with errors will be printed in addition to being displayed on the screen. This parameter has no meaning if the L or C options are given since listing produced under those options will include error flags anyway.

R The line numbers for referenced labels in an operand string will be printed at the right margin of the listing. The line number is the line on which the Referenced label was defined. If the L, C, or E option is not also given, this option has no effect. This option may be given instead of or in addition to the X option. The R option is especially convenient with GOTO or CALL instructions in following the logic path of a complex set of code. Note that for the R option to be effective, a printer with at least 130 column printing capability must be used.

No more than two labels may be printed on any single line. This means that, in some cases, it is necessary to use the C option in conjunction with the R option. Using the C option causes multiple listing lines per DATABUS instruction allowing more than three labels to be printed

X A cross-reference listing is printed at the end of the compilation. There will actually be two cross-referlences: one for the data labels and one for the executable labels. Each cross-reference is sorted alphabetically. The data or executable label is given

preceded by the octal location where the label was defined and followed by a list of all line numbers in which the item was defined or referenced. An asterisk flags those line numbers which are definitions. The SORT utility is called by the compiler to do the actual reference sorting, and the messages displayed on the screen will be appropriate to the progress of the sort. A cross-reference may be obtained regardless of whether a listing was requested.

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

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

HEADING:

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

Example:

```
DBCMP PROGRAM
```

This is the simplest compilation specification. The following items are pertinent:

-- The source code found in file PROGRAM/TXT will be compiled. All drives will be searched for PROGRAM/TXT starting with drive zero (0).

-- The object code will be placed in PROGRAM/DBC. The object code will be placed on the same drive as the source unless the object already exists on another drive.

-- No other output would be given except for errors displayed on the screen.

Example:

```
DBCMP ANSWER,ANSWER4;CX
```

The following items are pertinent:

-- The source code file ANSWER/TXT would be compiled.

-- All drives starting with drive zero (0) would be searched ANSWER/TXT.

-- The object code would be placed in ANSWER4/DBC on the same drive as ANSWER/TXT unless ANSWER4/DBC already exists on another drive.

-- A listing would be printed on the printer and would consist of the source and object code with a data and executable label cross-reference at the end.

Example:

```
DBCMP FILE:DR0,,FILELST/TXT:DR1;LX
```

The following items are pertinent:

-- The source code in FILE/TXT on drive zero (0) would be compiled.

-- The object code would be placed in FILE/DBC on drive zero (0).

-- A copy of the source code and a data and label cross-reference will be written in FILELST/TXT on drive one (1).

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

16.3 Compilation Diagnostics

The compiler prints and displays diagnostic messages on the listing to help the programmer debug snytactical errors in his code. These messages take the form of an error code letter at the left of the listing and an asterisk under the line at the position of the scanning pointer when the error occured. The letters and their meanings are:

D Duplicate label.

E Expression error. A generalized syntactical error.

I Unrecognizable instruction.

U Undefined variable or label.

In the case of E errors a number is given on the line with the asterisk pointing out the error position in the source line. This number refers to the list of detailed error explanations in Appendix E of this document. If any of these flags appear, the compiler will store a STOP instruction into the first executable location in the object file. If the faulty program is then executed, it will only execute the STOP instruction which will simply return control to the MASTER program.

The interpreter uses the DOS logical file zero (0) 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 and WRITE statement (since fetching the statement also involves a disk I/O). For this reason, the DOS DATABUS compiler will insert a TABPAGE instruction if it detects a READ or WRITE statement crossing a segment boundary. Normally, this is of no particular concern to the user, however, programs using TABPAGE and doing extensive optimization should be aware that this may occur.

16.4 Disk Space Requirements

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

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

completion the four (4) files, DSCREF/SYS, *SORTMRG/SYS, *SORTKEY/SYS, and DSCREFT/SYS will be deleted and the file space again made available to the user.

APPENDIX A. INSTRUCTION SUMMARY

SYNTACTIC DEFINITIONS

<label>	is a letter, followed by any combination of up to seven (7) letters and digits.
<string>	is any sequence of characters with the exceptions noted in section 2.6 (forcing character).
<svar>	A name assigned to a statement defining a character string variable.
<nvar>	A name assigned to a statment defining a numeric string variable.
<ssvar>	A name assigned to a statment defining a source character string variable. This variable is unchanged as a result of the instruction.
<dsvar>	A name assigned to a statement defining a destination character string variable. This variable is generally changed as a result of the instruction.
<snvar>	A name assigned to a statement defining a source numeric string variable. This variable is unchanged as a result of the instruction.
<dnvar>	A name assigned to a statement defining a destination numeric string variable. This variable is generally changed as a result of the instruction.
<slit>	is a literal of the form "<string>" (see section 2.5).
<nlit>	is a literal of the form "<string>" where string is a valid numeric string (see section 2.5).
<char>	is any single character of the form

	"<string>" where string is of length one (1).
<occ>	is an octal control character (001 to 037 inclusive).
<list>	Any combination of <slit>, <occ>, <list controls> (see section 9.1.3), <nvar> and <svar>.
<cmlist>	is a name assigned to a statement defining a COMLST data declaration.
<nlist>	A list of numeric variables each pair of which is separated by a comma (,). The list may be continued on more that one line by placing a colon (:) after the last variable on the line to be continued.
<slist>	A list of character string variables each pair of which is separated by a comma (,). The list may be continued on more than one line by placing a colon (:) after the last variable on the line of to be continued.
<nslist>	any combination of numeric and character string variables separated by commas. The list may be continued on more than one line by placing a colon (:) after the last variable on the line to be continued.
<blist>	The name assigned to the first of a set of physically contiguous numeric string or character string variables.
<index>	A numeric variable used in connection with list accessing.
<dnum>	A decimal number between -128 and 127.
<dnum1>	A decimal number indicating the number of digits that should precede the decimal point.
<dnum2>	A decimal number indicating the number of

	digits that should follow the decimal point.
<dnum3>	A decimal number between 1 and 20 inclusive.
<dnum4>	A decimal between 1 and 64 inclusive.
<flag>	One of the following flags: OVER, LESS, ZERO, or EOS (EQUAL and ZERO are two names for the same flag) that are used to indicate the result of some DATABUS operation.
<event>	The occurance of a program trap: PARITY, RANGE, FORMAT, CFAIL, or IO.
<skey>	A numeric or character string variable used with SEARCH.
<DOS file spec>	A DOS compatible file specification (see DOS user's guide).
<file>	A name assigned to a FILE declaration.
<rfile>	A name assigned to a RFILE declaration.
<ifile>	A name assigned to a IFILE declaration.
<rifile>	A name assigned to a RIFILE declaration.
<rn>	A numeric variable which contains a positive record number (>=0) used to randomly READ or WRITE a file.
<seq>	A numeric variable which contains a negative number (<0) used to READ or WRITE a file sequentially.
<key>	A non-null string variable used as a key to indexed I/O accesses.
<null>	A null string variable used as a key to an indexed read.
<route>	A character string variable used for routing.

FOR THE FOLLOWING SUMMARY:

Items enclosed in brackets [ ] are optional.

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

COMPILER DIRECTIVES

```
<label>  EQU       <dnum>
<label>  EQUATE    <dnum>
         INC       <DOS file spec>
         INCLUDE   <DOS file spec>
```

FILE DECLARATIONS

```
<label>  FILE
<label>  IFILE
<label>  RFILE
<label>  RIFILE
```

DATA DEFINITIONS

```
<label>  FORM      <dnum1>.<dnum2>
<label>  FORM      <dnum1>.
<label>  FORM      <dnum1>
<label>  FORM      <nlit>
<label>  DIM       <dnum>
<label>  INIT      <slit>
<label>  FORM      *<dnum1>.<dnum2>
<label>  FORM      *<dnum1>.
<label>  FORM      *<dnum1>
<label>  FORM      *<nlit>
<label>  DIM       *<dnum>
<label>  INIT      *<slit>
<label>  COMLST    <dnum4>
```

CONTROL

```
GOTO        <label>
GOTO        <label> IF <flag>
GOTO        <label> IF NOT <flag>
BRANCH      <index><prep><list>
CALL        <label>
CALL        <label> IF <flag>
CALL        <label> IF NOT <flag>
ACALL       <nslist>
RETURN
RETURN      IF <flag>
RETURN      IF NOT <flag>
STOP
STOP        IF <flag>
STOP        IF NOT <flag>
CHAIN       <svar>
CHAIN       <slit>
TRAP        <label> IF <event>
TRAPCLR     <event>
ROLLOUT     <svar>
ROLLOUT     <slit>
PI          <dnum3>
TABPAGE
DSCNCT
```

CHARACTER STRING HANDLING

MATCH	<svar><prep><svar>
MATCH	<slit><prep><svar>
MOVE	<ssvar><prep><dsvar>
MOVE	<sslit><prep><dsvar>
MOVE	<ssvar><prep><dnvar>
MOVE	<sslit><prep><dnvar>
MOVE	<snvar><prep><dsvar>
MOVE	<snlit><prep><dsvar>
APPEND	<ssvar><prep><dsvar>
APPEND	<sslit><prep><dsvar>
APPEND	<snvar><prep><dsvar>
CMOVE	<ssvar><prep><dsvar>
CMOVE	<char><prep><dsvar>
CMOVE	<occ><prep><dsvar>
CMATCH	<svar><prep><dvar>
CMATCH	<char><prep><dvar>
CMATCH	<svar><prep><char>
CMATCH	<svar><prep><occ>
CMATCH	<occ><prep><dvar>
BUMP	<dsvar>
BUMP	<dsvar><prep><dnum>
RESET	<dsvar><prep><dnum>
RESET	<dsvar><prep><ssvar>
RESET	<dsvar><prep><snvar>
RESET	<dsvar>
ENDSET	<dsvar>
LENSET	<dsvar>
CLEAR	<dsvar>
EXTEND	<dsvar>
LOAD	<dsvar><prep><index><prep><slist>
STORE	<ssvar><prep><index><prep><slist>
STORE	<sslit><prep><index><prep><slist>
CLOCK	TIME<prep><dsvar>
CLOCK	DAY<prep><dsvar>
CLOCK	YEAR<prep><dsvar>
TYPE	<svar>
SEARCH	<skey><prep><blist><prep><nvar><prep><dsvar>
REPLACE	<ssvar><prep><dsvar>
REP	<ssvar><prep><dsvar>
REPLACE	<sslit><prep><dsvar>
REP	<sslit><prep><dsvar>

ARITHMETIC

ADD	<snvar><prep><dnvar>
ADD	<nlit><prep><dnvar>
SUB	<snvar><prep><dnvar>
SUB	<nlit><prep><dnvar>
SUBTRACT	<snvar><prep><dnvar>
SUBTRACT	<nlit><prep><dnvar>
MULT	<snvar><prep><dnvar>
MULT	<nlit><prep><dnvar>
MULTIPLY	<snvar><prep><dnvar>
MULTIPLY	<nlit><prep><dnvar>
DIV	<snvar><prep><dnvar>
DIV	<nlit><prep><dnvar>
DIVIDE	<snvar><prep><dnvar>
DIVIDE	<nlit><prep><dnvar>
MOVE	<snvar><prep><dnvar>
MOVE	<nlit><prep><dnvar>
COMPARE	<nvar><prep><nvar>
COMPARE	<nlit><prep><nvar>
LOAD	<dnvar><prep><index><prep><nlist>
STORE	<snvar><prep><index><prep><nlist>
STORE	<nlit><prep><index><prep><nlist>
CHECK11	<svar><prep><svar>
CK11	<svar><prep><slit>
CHECK10	<svar><prep><svar>
CK10	<svar><prep><slit>

INPUT/OUTPUT

KEYIN	<list>[;]
DISPLAY	<list>[;]
CONSOLE	<list>[;]
BEEP	
PRINT	<list>[;]
RPRINT	<list>[;]
RELEASE	
PREPARE	<file¦rfile>,<svar¦slit>
PREP	<file¦rfile>,<svar¦slit>
OPEN	<file¦rfile¦ifile¦rifile>,<svar¦slit>
CLOSE	<file¦rfile¦ifile¦rifile>
WRITE	<file¦rfile>,<rn¦seq>;<;¦<list>[;]>
WRITE	<ifile¦rifile>,<rn¦seq¦key>;<;¦<list>[;]>
WRITAB	<file¦rfile>,<rn¦seq>;<;¦<list>[;]>
WEOF	<file¦rfile¦ifile¦rifile>,<rn¦seq>
UPDATE	<ifile¦rifile>;<;¦<list>[;]>
READ	<file¦rfile>,<rn¦seq>;<;¦<list>[;]>

```
READ      <ifile¦rifile>,<rn¦seq¦key¦null>;<;¦<list>[;]>
READKS    <ifile¦rifile>;<;¦<list>[;]>
DELETE    <ifile¦rifile>,<key>
INSERT    <ifile¦rifile>,<key>
COMCLR    <cmlist>
COMTST    <cmlist>
COMWAIT
SEND      <cmlist>,<route>;<nslist>
RECV      <cmlist>,<route>;<slist>
DEBUG
```

APPENDIX B. INPUT/OUTPUT LIST CONTROLS

CONTROL	USED IN	FUNCTION
*P<h>:<v>	KDC	Causes the cursor to be positioned horizontally and vertically to the column and line indicated by the numbers <h> (horizontal 1-80) and <v> (vertical 1-24). These numbers may either be literals or numeric variables. Note that <v> is ignored in the CONSOLE statement. This list control is only effective on the Datapoint 3502 (see 9.1.3.1, 9.2.3.1, and 9.3).
*N	KDPR	Causes the cursor or printer to be positioned in Column 1 of the next line (see 9.1.3.7, 9.2.3.7, 10.1.3.4 and 10.2).
*EL	KDC	Causes the line to be erased from the current cursor position (see 9.1.3.2, 9.2.3.2, and 9.3).
*EF	KDC	Causes the screen to be erased from the current cursor position (see 9.1.3.3, 9.2.3.3, and 9.3).
*ES	KD	Causes the cursor to be positioned at horizontal position 1 of the top row of the display and the entire display to be erased (see 9.1.3.4 and 9.2.3.4).
*EOFF	K	Causes the echo during input operations from the terminal to be defeated (see 9.1.3.13).
*EON	K	Causes the echo during input operations from the terminal to be on (see 9.1.3.14).
*+	KDCPR	Turn on Keyin Continuous for KEYIN or space after logical length suppression for DISPLAY, PRINT, RPRINT, and CONSOLE (see 9.1.3.9, 9.2.3.9, 9.3, 10.1.3.8, and 10.2).

*+	W	Turn on space compression during WRITE (see section 12.3.4.3.1).
*-	KDCPR	Turn off Keyin Continuous (turned off at the end of the statement) or the space after logical length suppression (see 9.1.3.10, 9.2.3.10, 9.3, 10.1.3.9, and 10.2).
*-	W	Turn off space compression during WRITE (see section 12.3.4.3.2).
*<n>	PR	Causes a horizontal tab on the printer to the column indicated by the number <n> (see 10.1.3.5 and 10.2).
*<n>	RW	Tab specification for READ or WRITAB operations.
*<nvar>		The logical file pointers are moved to that character position relative to the current physical record (see section 13.4.1 and section 13.6.1).
;	KDPR	Suppress a new line function when occurring at the end of a list (see 9., 10.1.3.6 and 10.2)
*F	PR	Causes the printer to be positioned to the top of form (see 10.1.3.1 and 10.2).
*L	KDPR	Causes a linefeed to be displayed or printed (see 9.1.3.6, 9.2.3.6, 10.1.3.3 and 10.2).
*C	KDPR	Causes a carriage return to be displayed or printed (see 9.1.3.5, 9.2.3.5, 10.1.3.2 and 10.2).
*T	K	Time out after 2 seconds for KEYIN statement (see 9.1.3.11).
*W	KD	Pause for one second (see 9.1.3.12 and 9.2.3.12).
*JL	K	Left justify numeric variable and zero fill at right if there is no decimal point. Left justify string variable and

		blank (or zero if *ZF option is given) fill to ETX (see 9.1.3.17).
*JR	K	Right justify string variable and blank (or zero if *ZF option is given) fill at left (see 9.1.3.18).
*ZF	K	Zero fill instead of blank fill string variable (see 9.1.3.19).
*ZF	PRW	Left zero fill numeric variable (see sections 10.1.3.7, 10.2 and 12.3.4.3.3).
*DE	K	Restrict string input to digits (0-9) only (see 9.1.3.20).
*IT	K	Set Text-input mode. This applies to 3601 only (see 9.1.3.15).
*IN	K	Clear Text-input mode. This applies to 3601 only (see 9.1.3.16).
*MP	W	Convert numeric variable to minus overpunch format (see section 12.3.4.3.4).

APPENDIX C. SAMPLE DATASHARE SYSTEM

The programs described in the following sections are a complete set of the programs necessary to bring up a DATASHARE system. This system includes a method of logging activity on the system and a great deal of system security.

The following is list of the events that are logged by the system: user sign on's, user sign off's, invalid attempts to sign on, all program errors not controlled by the user's program, all attempts to execute a program, all attempts to rollout and all rollout returns.

System security is provided by requiring that a user have valid identification before allowing him to sign on. Additional security is provided by assigning security clearances to all users, then requiring the appropriate security clearance before allowing a user to execute a program.

** SPECIAL NOTE **

The source files for the programs described in this Appendix are included on the released object tape. These source files are provided solely for the customer's convenience. They are <u>not</u> a part of the supported software. Any errors or suggested modifications to these programs should be submitted as a USER'S COMMENT on this user's guide.

To generate this system:

-- use the DOS MIN command to transfer the following programs to your disk,

-- use the chain file provided (see section C.1.5.1) and the DOS CHAIN command to compile the system programs (a description of the chain file tags to be used is included in the chain file). A suggested DOS command line to compile the system programs is:

    CHAIN MAKEANMA/CHN;1,2,3,4,SAMPLE,NEW

-- compile the supplemental system program "NEWUSER" (see section C.3.1),

-- execute the DATASHARE interpreter (see the user's guide of the

appropriate interpreter),

-- when the ANSWER program asks:

What is your identification number?

you should type: 999999999

This will sign you onto the system as "Anyuser" with the highest possible security clearance. (Note: for added security, identification numbers are not displayed on the screen.)

-- When the MASter MENU program asks for a program number by displaying:

Selection by number __

you should type: 3

to get "Program Selection by Name".

-- In response to:

ENTER PROGRAM NAME:
you should type: NEWUSER

to execute the NEWUSER program.

-- When the NEWUSER program asks for a program number as follows:

Selection by number __

you should type: 1

to "Authorize a new user".

-- In response to:

_____ _____ _
Enter the user's identification number.

you should type your social security number or any other 9-digit number you want to use as your identification number.

-- In response to the successive requests:

Enter the user's name.

Enter the user's security clearance.

you should type your name followed by a 9. Note: this assumes you are the system engineer and will be one of the few people who will have the highest possible security clearance.

-- When the NEWUSER program asks for another user's identification number, you should indicate no more additions by tapping the ENTER key.

-- In response to:

Are you done? (Y/N)

you should type: Y

to indicate that you are done.

-- At this point you are returned to the NEWUSER menu. You should now either delete "Anyuser" from the list of authorized users or modify his security clearance to one of the lowest possible levels. (Remember: "Anyuser's" identification number is 999999999.)

-- Now you should type: 99

to allow the NEWUSER program to continue.

-- To add any more users to the list of authorized user's you may use the NEWUSER program.

C.1 SYSTEM PROGRAMS

The following programs must be compiled to initiate the execution of this DATASHARE system.

C.1.1 Sample ANSWER Program

```
. DATASHARE ANSWER PROGRAM
.
. NOTE:  THE PORT NUMBER INCLUSION FILE IS NAMED "PORTN/TXX" TO
.        DEMONSTRATE THAT EXTENSIONS OTHER THAN "/TXT" MAY BE
.        USED FOR INCLUSION FILES.
.
PORTN     INCLUDE   PORTN/TXX           (PORTN FORM " 1", ETC.)
TODAY     INIT      "mm/dd/yy"          DATE PASSED IN COMMON
*.....................................................................
SECURITY FORM       1                   SEE DESCRIPTION BELOW
.
. THIS VARIABLE IS USED TO INDICATE THE SECURITY RATING FOR WHICH
. THE USER IS AUTHORIZED.  IT IS INITIALIZED FROM THE FILE OF
. AUTHORIZED USERS.  A PROGRAM CAN REQUIRE THAT THE USER HAVE THE
. SECURITY CLEARANCE NECESSARY TO USE THAT PROGRAM.  ZERO IS THE
. LOWEST RATING AND NINE IS THE HIGHEST RATING.
* EXAMPLE:
. SAY THAT THE USE OF PROGRAM "ROLLOUT" IS TO BE RESTRICTED TO
. ONLY A FEW USERS.  THOSE USERS WHO WILL BE ALLOWED TO USE
. "ROLLOUT" WILL BE GIVEN A SECURITY RATING OF 7.  ALL OTHER
. USERS WILL BE GIVEN LOWER SECURITY RATINGS.  PROGRAM "ROLLOUT"
. IS THEN WRITTEN SO THAT IT WILL "ROLLOUT" ONLY WHEN THE USERS
. RATING IS 7.  IF THE USER ATTEMPTING TO USE "ROLLOUT" HAS ANY
. RATING LESS THAN 7, THEN A STOP INSTRUCTION IS EXECUTED.
.
* SOME SUGGESTED SECURITY LEVELS ARE AS FOLLOWS:
.
.     SECURITY   USED BY:
.        0 ..... DISCONTINUED SERVICE
.        1-3 ... DATA ENTRY OPERATORS
.        4-6 ... DATA ENTRY SUPERVISORS
.        7 ..... PROGRAMMER
.        8 ..... PROJECT MANAGER
.        9 ..... SYSTEMS PROGRAMMER
*.....................................................................
. SCRATCH VARIABLES
.
CWK2      DIM       2
CWK3      DIM       3
NWK2      FORM      2
NWK9      FORM      9
```

```
*.....................................................................
USERS    IFILE                          FILE OF AUTHORIZED USERS
USERID   DIM       9                    KEY USED WITH FILE "USERS"
*.....................................................................
ROLCHAIN FILE                           ROLLOUT CHAIN FILE
SEQ      FORM      "-1"
*.....................................................................
. OUTPUT PARAMETERS OF SUBROUTINE "GETDATE"
.                                       TIME IN 24-HR. FORMAT
TIME     INIT      "hh:mm:ss"           hours:minutes:seconds
DAY      FORM      2
MONTH    FORM      2                    mm/dd/yy
YEAR     FORM      2
*
JDAY     FORM      3                    JULIAN DATE
CDAY     DIM       3                    DAY (CHARACTER STRING)
CYEAR    DIM       2                    YEAR (CHARACTER STRING)
*
NFEB     FORM      "28"                 # OF DAYS IN FEBRUARY
N30      FORM      "30"                 USED FOR 30-DAY MONTHS
N31      FORM      "31"                 USED FOR 31-DAY MONTHS
*.....................................................................
NAME     DIM       20                   USER'S NAME FROM LOG FILE
         INCLUDE   LOGDATA/TXT
```

```
+.....................................................................
. MAINLINE
.
        INCLUDE   LOGIO/TXT
*.....................................................................
. SET THE TRAPS SO THAT IF ANYTHING GOES WRONG THE USER STILL
. CANNOT LOG ON WITHOUT AUTHORIZATION
.
        TRAP      BADANS IF IO
        TRAP      BADANS IF CFAIL
        TRAP      BADANS IF FORMAT
        TRAP      BADANS IF RANGE
        TRAP      BADANS IF PARITY
*.....................................................................
. LOG THE USER OFF THE SYSTEM.
.
. THIS FUNCTION IS PUT AT THIS POINT IN THE ANSWER PROGRAM SO
. THAT ANYONE WHO TURNS OFF THE PORT (INSTEAD OF USING A NORMAL
. LOG-OFF) WILL STILL GET LOGGED-OFF.  NOTE THAT; WHEN THE PORT
. IS TURNED OFF, THE ANSWER PROGRAM WILL CONTINUE EXECUTING UNTIL
. THE FIRST CONSOLE, KEYIN OR DISPLAY STATEMENT IS REACHED.
.
*.....................................................................
. GET THE DATE AND TIME.
.
        TRAP      BADANS IF IO
        CALL      GETDATE
        MOVE      "LOG-OFF" TO LOGTYPE
        CLEAR     LOGINFO              WRITES BLANK "OTHER INFO."
        CALL      LOGWRITE
*.....................................................................
. OPEN THE FILE OF AUTHORIZED USERS
.
. NOTE:  THE NAME OF THE INDEX FILE NEED NOT BE THE SAME AS THE
.        NAME OF THE TEXT FILE WHICH IS INDEXED.  (FOR INSTANCE;
.        "USERS/ISI" COULD BE CREATED FROM A TEXT FILE NAMED
.        "USERS/DSP".)  THIS PROVIDES ADDITIONAL SYSTEM SECURITY,
.        SINCE "USERS/DSP" CANNOT BE ACCESSED BY DATABUS PROGRAMS
.
        TRAP      NOUSER IF IO
        OPEN      USERS,"USERS"
        TRAP      BADANS IF IO
        GOTO      LOGON
```

```
*.....................................................................
. FILE CONTAINING AUTHORIZATIONS IS MISSING
.
NOUSER   DISPLAY   *ES:
                   *P20:4,"You cannot log-on, the file contain-":
                   *P20:5,"ing the list of authorized users is":
                   *P20:6,"missing!  To use the system, you":
                   *P20:7,"must use the DOS INDEX utility to":
                   *P20:8,"create the file, #"USERS/ISI#"."
         GOTO      HANG
*.....................................................................
. LOG THE USER ONTO THE SYSTEM
.
LOGON    DISPLAY   *ES,*P15:4,"D A T A S H A R E   S Y S T E M ":
                                 "  O N - L I N E":
                   *P30:6,"You are on port ##",PORTN
*.....................................................................
. DISPLAY THE DATE AND TIME
.
         CALL      GETDATE
         DISPLAY   *P31:8,"Today is ",TODAY
*.....................................................................
. CHECK TO SEE IF THE USER IS AUTHORIZED
.
. NOTE:  FOR ADDITIONAL SECURITY, ECHO OFF IS USED WHILE ENTERING
.        THE IDENTIFICATION NUMBER.  THIS PREVENTS THE ID FROM
.        BEING DISPLAYED AT THE PORT.
.
.        THE IDENTIFICATION NUMBER BEING REQUESTED IS THE USERS
.        SOCIAL SECURITY NUMBER.  OTHER IDENTIFICATION TECHNIQUES
.        MAY BE USED.
.
         KEYIN     *P1:12,"What is your identification number? ":
                   *EL,*EOFF,NWK9
         COMPARE   "100000000" TO NWK9       MAKE SURE HE ENTERS
         GOTO      BADID IF LESS             ALL 9 DIGITS
         MOVE      NWK9 TO USERID
         GOTO      READID
```

```
*.....................................................................
. UN-AUTHORIZED USER
.
. NOTE:  THE PROGRAMMER CAN SET THE PENALTY FOR ENTERING A BAD
.        ID BY ADJUSTING THE NUMBER OF *W'S USED
.
BADID    BEEP
         DISPLAY   *P50:12,*EL,"You are not an authorized user!":
                   *W,*W,*W,*W,*W
         BEEP
         MOVE      "BAD ID" TO LOGTYPE
         MOVE      NWK9 TO LOGINFO
         CALL      LOGWRITE
         TRAP      BADANS IF IO
         GOTO      LOGON
*.....................................................................
. SEE IF THE USER IS AUTHORIZED
.
READID   READ      USERS,USERID;USERID,NAME,SECURITY
         GOTO      BADID IF OVER              CAN'T FIND HIS NUMBER
         DISPLAY   *P50:12,*EL,"Thank you",*W,*W,*P1:12,*EL
         MOVE      "LOG-ON" TO LOGTYPE
         MOVE      NAME TO LOGINFO
         CALL      LOGWRITE
         TRAP      BADANS IF IO
         DISPLAY   *P20:10,"Hello, ",NAME:
                   *P20:11,"You are logged on at ",TIME,".":
                   *W,*W,*W
*.....................................................................
. IF USER HAS HIGH ENOUGH SECURITY CLEARANCE, CHECK TO SEE IF
. LOG FILE NEEDS CLEANING
.
         COMPARE   "4" TO SECURITY
         STOP      IF LESS              "CHAIN to MASTER"
*.....................................................................
. LOOK AT THE NUMBER OF LOG ENTRIES
. IF MORE THAN 500, TELL THE USER HE NEEDS TO REORGANIZE
.
         COMPARE   "500" TO LOGRN
         STOP      IF LESS              "CHAIN to MASTER"
```

```
*.....................................................................
. FIND OUT IF THE USER WANTS TO RE-ORGANIZE THE LOG FILE
.
         KEYIN     *ES:
                   *P20:4,"The log file is now using more than":
                   *P20:5,"five hundred disk sectors.  It needs":
                   *P20:6,"to be re-organized to free this":
                   *P20:7,"space.":
                   *P1:12,"Do you want to re-organize the log ":
                   "file? (Y/N) ",CWK3
         CMATCH    "Y" TO CWK3
         STOP      IF NOT EQUAL        "CHAIN to MASTER"
*.....................................................................
. RE-ORGANIZE THE LOG FILE
. CHAIN FILE NEEDS TO BE WRITTEN SO OPEN CHAIN FILE
.
         DISPLAY   *ES,"Writing CHAIN file."
         MOVE      "ROLLOUT" TO LOGTYPE
         MOVE      "RE-ORGANIZE LOG FILE" TO LOGINFO
         CALL      LOGWRITE
         TRAP      NOCHAIN IF IO
         PREPARE   ROLCHAIN,"ROLCHAIN"
         TRAP      BADANS IF IO
         GOTO      WRITECHN
*.....................................................................
. CHAIN FILE COULD NOT BE CREATED
.
NOCHAIN  DISPLAY   *ES:
                   *P20:4,"CHAIN file could not be written!":
                   *P20:5,"Re-organization discontinued.":
                   *W,*W
         WRITAB    LOGFILE,LOGRN;*12,"NO ROLLOUT"
         STOP                          "CHAIN to MASTER"
*.....................................................................
. WRITE THE CHAIN FILE
.
WRITECHN WRITE     ROLCHAIN,SEQ;*+,". RE-ORGANIZE SYSTEM LOG FILE
         WRITE     ROLCHAIN,SEQ;"."
         WRITE     ROLCHAIN,SEQ;"//."
         WRITE     ROLCHAIN,SEQ;"//* TAP THE DISPLAY KEY TO ":
                                "START RE-ORGANIZATION"
         WRITE     ROLCHAIN,SEQ;"//."
         WRITE     ROLCHAIN,SEQ;"//. SAVE THE LOG FILE"
         WRITE     ROLCHAIN,SEQ;"//."
```

```
*
        WRITE     ROLCHAIN,SEQ;". Either of the following ":
                              "techniques may by used:"
        WRITE     ROLCHAIN,SEQ;"."
        WRITE     ROLCHAIN,SEQ;". SAPP MASTERLG,LOGFILE,":
                              "MASTERLG"
        WRITE     ROLCHAIN,SEQ;".      or"
        WRITE     ROLCHAIN,SEQ;". LIST LOGFILE;L"
        WRITE     ROLCHAIN,SEQ;". LISTING OF MASTER LOG FILE"
*
        WRITE     ROLCHAIN,SEQ;"LIST LOGFILE;L"
        WRITE     ROLCHAIN,SEQ;"LISTING OF MASTER LOG FILE"
        WRITE     ROLCHAIN,SEQ;"//."
        WRITE     ROLCHAIN,SEQ;"//. RE-CREATE THE LOG FILE"
        WRITE     ROLCHAIN,SEQ;"//."
        WRITE     ROLCHAIN,SEQ;"BUILD LOGFILE;!"
        WRITE     ROLCHAIN,SEQ;"*000":
                               "  PORT":
                               "  LOG TYPE":
                               "     DATE":
                               "      TIME":
                               "   OTHER INFORMATION"
        WRITE     ROLCHAIN,SEQ;"*rec":
                               "   ()":
                               "  (         )":
                               "  (       )":
                               "  (       )":
                               "  ( . . ."
        WRITE     ROLCHAIN,SEQ;"!"
*
        WRITE     ROLCHAIN,SEQ;"//."
        WRITE     ROLCHAIN,SEQ;"//. RETURN TO DATASHARE"
        WRITE     ROLCHAIN,SEQ;"//."
        WRITE     ROLCHAIN,SEQ;"DSBACKTD"
        WEOF      ROLCHAIN,SEQ
*.....................................................
. ROLLOUT TO THE CHAIN FILE
. CHAIN TO THE MASTER MENU
.
        DISPLAY   *ES,"Re-organization in progress."
        TRAP      NOCHAIN IF CFAIL
        ROLLOUT   "CHAIN ROLCHAIN"
        TRAP      BADANS IF CFAIL
        MOVE      "ROLL RET" TO LOGTYPE
        CLEAR     LOGINFO
        CALL      LOGWRITE
        STOP                          "CHAIN to MASTER"
```

```
*.....................................................................
. SUBROUTINE TO GET THE TIME, DAY AND YEAR
.
. ON EXIT VARIABLE:  TIME = "hr:mn:sc"
.                    DAY = "dd"
.                    MONTH = "mm"
.                    YEAR = "yy"
.                    TODAY = "mm/dd/yy"
.
GETDATE  CLOCK     YEAR TO CYEAR        GET THE YEAR
         CLOCK     DAY TO CDAY          GET THE DAY
         CLOCK     TIME TO TIME         GET THE TIME
*.....................................................................
. PERFORM BOUNDARY CONDITION CHECKS IF DESIRED
.
.        CLOCK     DAY TO CWK3             IF TIME TAKEN BEFORE
.        MATCH     CDAY TO CWK3            MIDNIGHT AND DAY TAKEN
.        GOTO      GETDATE IF NOT EQUAL    AFTER MIDNIGHT, REPEAT
*
.        CLOCK     YEAR TO CWK2            IF DAY TAKEN BEFORE
.        MATCH     CYEAR TO CWK2           NEW YEARS & YEAR TAKEN
.        GOTO      GETDATE IF NOT EQUAL    AFTER NEW YEARS, REPEAT
*.....................................................................
         MOVE      CDAY TO JDAY
         MOVE      "0" TO MONTH         INITIALIZE
         MOVE      CYEAR TO YEAR
*
         COMPARE   "1" TO JDAY          SYSTEM INITIALIZED
         GOTO      NODATE IF LESS       WITHOUT DATE!
*.....................................................................
. PERFORM YEAR-CHECK IF DESIRED
.
.        COMPARE   "70" TO YEAR
.        GOTO      NODATE IF LESS
.        COMPARE   "80" TO YEAR
.        GOTO      NODATE IF NOT LESS
*.....................................................................
. MAKE SURE FEBRUARY IS HANDLED PROPERLY ON LEAP YEARS
.
         MOVE      YEAR TO NWK2
         DIVIDE    "4" INTO NWK2
         MULTIPLY  "4" INTO NWK2
         COMPARE   YEAR TO NWK2            IS IT A LEAP YEAR?
         GOTO      MDLOOP IF NOT EQUAL     NO, LEAVE NFEB = 28.
         MOVE      "29" TO NFEB            YES, SET NFEB = 29.
```

```
*.....................................................
. COMPUTE THE MONTH
.
MDLOOP    ADD       "1" TO MONTH
          LOAD      NWK2 FROM MONTH OF N31,NFEB,N31:   JAN/FEB/MAR
                                       N30,N31,N30:    APR/MAY/JUN
                                       N31,N31,N30:    JUL/AUG/SEP
                                       N31,N30,N31     OCT/NOV/DEC
          SUBTRACT  NWK2 FROM JDAY
          GOTO      MDL1 IF EQUAL       SUBTRACT DAYS OF THE MONTH
          GOTO      MDLOOP IF NOT LESS  UNTIL MONTH FOUND
*
MDL1      ADD       NWK2 TO JDAY        UNBIAS FROM LAST SUBTRACT
          MOVE      JDAY TO DAY         TO GET DAY OF THE MONTH
*.....................................................
. PUT THE DATE INTO mm/dd/yy FORMAT
.
MDL2      CLEAR     TODAY
          APPEND    MONTH TO TODAY
          APPEND    "/" TO TODAY
          MOVE      DAY TO CWK2
          CMATCH    " " TO CWK2         IS THERE A LEADING BLANK?
          GOTO      MDL3 IF NOT EQUAL   NO, CONTINUE
          CMOVE     "0" TO CWK2         YES, REPLACE IT WITH 0
MDL3      APPEND    CWK2 TO TODAY
          APPEND    "/" TO TODAY
          MOVE      YEAR TO CWK2
          CMATCH    " " TO CWK2         IS THERE A LEADING BLANK?
          GOTO      MDL4 IF NOT EQUAL   NO, CONTINUE
          CMOVE     "0" TO CWK2         YES, REPLACE IT WITH 0
MDL4      APPEND    CWK2 TO TODAY
          RESET     TODAY
          RETURN
*.....................................................
. DATE IMPROPER OR NOT INITIALIZED
.
NODATE    BEEP
          KEYIN     *P1:8,*EF,"What is the current month? ",MONTH:
                    *N,"What is the current day? ",DAY:
                    *N,"What is the current year? ",YEAR
*.....................................................
. CHECK FOR INVALID DAY ENTERED
.
          COMPARE   "1" TO DAY
          GOTO      NODATE IF LESS
          COMPARE   "32" TO DAY
          GOTO      NODATE IF NOT LESS
```

```
*.....................................................................
. CHECK FOR INVALID MONTH ENTERED
.
         COMPARE    "1" TO MONTH
         GOTO       NODATE IF LESS
         COMPARE    "13" TO MONTH
         GOTO       NODATE IF NOT LESS
*.....................................................................
. CHECK FOR INVALID YEAR IF DESIRED
.
.        COMPARE    "70" TO YEAR
.        GOTO       NODATE IF LESS
.        COMPARE    "80" TO YEAR
.        GOTO       NODATE IF NOT LESS
         DISPLAY    *P1:12,"Thank you",*W,*W,*P1:8,*EF
         GOTO       MDL2
*.....................................................................
. A TRAP HAS OCCURED WHILE IN THE ANSWER PROGRAM.  DO NOT ALLOW
. A CHAIN TO THE MASTER PROGRAM
.
BADANS   DISPLAY    *P58:1,*EL,"  Unrecoverable system":
                    *P58:2,*EL,"  error!  Consult your":
                    *P58:3,*EL,"  programmer."
         GOTO       HANG
```

C.1.2 Sample MASTER Program

```
. DATASHARE MASTER PROGRAM FOR LOGGING ERRORS
.
*.....................................................
. COMMON AREA
. THIS AREA GETS OVERWRITTEN WITH AN 11-BYTE CHARACTER STRING
. VARIABLE WHEN AN ERROR OCCURS
.
. NOTE: "ERROR" USES THE SAME NUMBER OF BYTES OF USERS DATA AREA
.       AS THE VARIABLES "PORTN" AND "TODAY" DEFINED IN COMMON
.
ERROR    DIM       *12                 ERROR MESSAGE
SECURITY FORM      *1                  USER'S SECURITY CLEARANCE
*.....................................................
. NOTE:  THE PORT NUMBER INCLUSION FILE IS NAMED "PORTN/TXX" TO
.        DEMONSTRATE THAT EXTENSIONS OTHER THAN "/TXT" MAY BE
.        USED FOR INCLUSION FILES.
.
         INCLUDE   PORTN/TXX
TODAY    INIT      "  /  /  "
*.....................................................
ANSWER   DIM       8
TIME     INIT      "hh:mm:ss"          hours:minutes:seconds
CWK1     DIM       1                   WORK AREA: CHAR.TYPE,LEN=1
CWK2     DIM       2                   WORK AREA: CHAR.TYPE,LEN=2
CWK11    INIT      "          "        CHARACTER, LENGTH 11
         INCLUDE   LOGDATA/TXT
*.....................................................
. SEE IF THERE ARE ANY DATASHARE ERRORS.
. IF NO ERROR OCCURED, THE 2-BYTE PORT NUMBER WILL BE MOVED INTO
. THE WORK AREA.  IN THIS CASE, THE 9TH CHARACTER OF CWK11 WILL
. STILL BE A BLANK.
. IF AN ERROR OCCURED, THE 11-BYTE ERROR MESSAGE WILL BE MOVED
. INTO CWK11.  IN THIS CASE, THE 9TH CHARACTER OF CWK11 WILL BE
. AN ASTERISK.
. BY CHECKING THE 9TH CHARACTER, IT CAN BE DETERMINED WHETHER AN
. ERROR OCCURED OR NOT.
.
         MOVE      ERROR TO CWK11
         RESET     CWK11 TO 9
         CMATCH    "*" TO CWK11
         GOTO      MASMENU IF NOT EQUAL
         INCLUDE   LOGIO/TXT
```

```
*.....................................................................
. SINCE THE DATE PASSED IN COMMON HAS BEEN OVERWRITTEN, GET THE
. JULIAN DATE AND USE THAT FOR LOGGING
.
         CLOCK     DAY TO TODAY
         ENDSET    TODAY
         APPEND    "/" TO TODAY
         CLOCK     YEAR TO CWK2
         APPEND    CWK2 TO TODAY
         RESET     TODAY
*.....................................................................
. WRITE THE LOG-OFF
.
         CLOCK     TIME TO TIME
         MOVE      "ERROR" TO LOGTYPE
         MOVE      ERROR TO LOGINFO
         CALL      LOGWRITE
*.....................................................................
. GIVE THE USER A CHANCE TO LOOK AT THE SCREEN BEFORE ABORTING
.
         BEEP
         DISPLAY   *P1:1,*EL,"Untrapped DATASHARE error at ",TIME
         KEYIN     *P67:1,"(P)ause? ",*T,*+,CWK1
         CMATCH    "P" TO CWK1          CHECK FOR NULL STRING
         GOTO      LOGOFF IF NOT EQUAL
PAUSE    KEYIN     *P67:1,"(C)ontinue? ",*+,*EL,CWK1
         CMATCH    "C" TO CWK1
         GOTO      PAUSE IF NOT EQUAL
*.....................................................................
. CHAIN TO THE APPROPRIATE ANSWER PROGRAM
.
LOGOFF   MOVE      PORTN TO CWK2          GET THE PORT NUMBER
         COMPARE   "10" TO PORTN          REMOVE LEADING SPACES
         GOTO      BUILDANS IF NOT LESS
         RESET     CWK2 TO 2
*
BUILDANS CLEAR     ANSWER                 BUILD THE NAME
         APPEND    "ANSWER" TO ANSWER     FORM: ANSWERn
         APPEND    CWK2 TO ANSWER         WHERE: n IS THE PORT
         RESET     ANSWER                 NUMBER (0 < n < 17)
         TRAP      BADANS IF CFAIL
         CHAIN     ANSWER
```

```
*.....................................................................
. ANSWER PROGRAM COULD NOT BE FOUND
.
BADANS   DISPLAY   *P58:1,*EL,"  The system program":
                   *P58:2,*EL,"  #"",*+,ANSWER,"#" could not":
                   *P58:3,*EL,"  be found!  Consult":
                   *P58:4,*EL,"  your programmer."
         GOTO      HANG
*.....................................................................
. CHAIN TO THE MASTER MENU
.
         TRAP      BADMASM IF CFAIL
MASMENU  CHAIN     "MASMENU"
*.....................................................................
. THE MASTER MENU COULD NOT BE FOUND
.
BADMASM  DISPLAY   *P58:1,"  The system program":
                   *P58:2,"  #"MASMENU#" could not":
                   *P58:3,"  found!  Consult":
                   *P58:4,"  your programmer."
         GOTO      HANG
```

C.1.3 Sample DATASHARE MASter MENU

```
. MASMENU - DATASHARE MASTER MENU
.
. THIS PROGRAM WAS GENERATED USING THE "MAKEMENU" PROGRAM
. THEN MODIFIED WITH THE DOS "EDIT" COMMAND
.
. COMPILING THIS PROGRAM REQUIRES THAT THE FILES:  "COMMON/TXT",
. "LOGDATA/TXT" AND "LOGIO/TXT" EXIST ON ANY DRIVE WHICH IS ON-
. LINE.  THESE INCLUSION FILES CONTAIN THE INFORMATION COMMON TO
. ALL OF THE SYSTEM PROGRAMS.
.
         INCLUDE   COMMON/TXT
INDEX    FORM      2                    USER SELECTION VARIABLE
TIME     INIT      "hh:mm:ss"           hours:minutes:seconds
PROGRAM  DIM       9                    PROGRAM SELECTION VARIABLE
CWK2     DIM       2                    WORK VARIABLE
         INCLUDE   LOGDATA/TXT
```

```
+.....................................................................
. MAINLINE
.
         INCLUDE   LOGIO/TXT
*.....................................................................
. DISPLAY THE MENU
.
SHOWMENU DISPLAY   *ES:
                   "DATASHARE MASTER MENU":
                   *P51:1,"Today is ",TODAY:
                   *P01:03,"( 1) ":
                   "Payroll Menu":
                   *P01:04,"( 2) ":
                   "Exit to DOS":
                   *P01:05,"( 3) ":
                   "Program Selection by Name":
                   *EL
*.....................................................................
. GET THE PROGRAM'S INDEX
.
GETINDEX KEYIN     *P1:12,*EL,"Selection by number":
                   *P41:12,"Enter (99) when you are done.":
                   *P25:12,"__",*P25:12,INDEX
         COMPARE   "1" TO INDEX
         GOTO      GETINDEX IF LESS
         COMPARE   "99" TO INDEX
         GOTO      LOGOFF IF EQUAL
         COMPARE   "04" TO INDEX
         GOTO      GETINDEX IF NOT LESS
*.....................................................................
. BRANCH TO THE ROUTINE INDICATED BY THE INDEX
.
         TRAP      BADCHAIN IF CFAIL
         CLOCK     TIME TO TIME
         BRANCH    INDEX OF MENU1:  Payroll Menu
                            DOS:  Exit to DOS
                            OTHER   Program Selection by Name
         GOTO      GETINDEX
*.....................................................................
. PROGRAM DOES NOT EXIST.
.
BADCHAIN RETURN
*.....................................................................
. CHAIN INSTRUCTIONS
```

```
*.....................................................................
. Payroll Menu
.
MENU1     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "MENU1    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "MENU1"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. EXIT TO DOS REQUIRES SECURITY CLEARANCE
.
DOS       COMPARE   "6" TO SECURITY
          GOTO      GETINDEX IF LESS
*
          TRAP      NOROLL IF CFAIL
          MOVE      "ROLLOUT" TO LOGTYPE
          CLEAR     LOGINFO
          CALL      LOGWRITE            EXIT TO DOS BY EXECUTING
          ROLLOUT   "FREE"              THE DOS "FREE" COMMAND
*
          MOVE      "ROLL RET" TO LOGTYPE
          CLEAR     LOGINFO
          CALL      LOGWRITE
          GOTO      GETINDEX
*
NOROLL    WRITAB    LOGFILE,LOGRN;*12,"NO ROLLOUT"
          RETURN
*.....................................................................
. PROGRAM SELECTION BY NAME REQUIRES SECURITY CLEARANCE
.
OTHER     COMPARE   "7" TO SECURITY
          GOTO      GETINDEX IF LESS
*
GETPROG   KEYIN     *ES,"ENTER PROGRAM NAME: ",PROGRAM;
          MOVE      "PROGRAM" TO LOGTYPE
          MOVE      PROGRAM TO LOGINFO
          CALL      LOGWRITE
*.....................................................................
. DO NOT ALLOW HIM TO CHAIN TO OTHER MASTER OR ANSWER PROGRAMS
.
          MATCH     "MASTER" TO PROGRAM
          GOTO      BADPROG IF EQUAL
          MATCH     "ANSWER" TO PROGRAM
          GOTO      BADPROG IF EQUAL
          TRAP      BADPROG IF CFAIL
          CHAIN     PROGRAM
```

```
*.....................................................................
. PROGRAM DOESN'T EXIST
.
BADPROG  DISPLAY   "  <-- THAT PROGRAM DOES NOT EXIST!":
                   *W,*W
         WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
         GOTO      SHOWMENU
*.....................................................................
. LOG OFF BY CHAINING TO THE APPROPRIATE ANSWER PROGRAM
.
LOGOFF   MOVE      PORTN TO CWK2          GET THE PORT NUMBER
         COMPARE   "10" TO PORTN          REMOVE LEADING SPACES
         GOTO      BUILDANS IF NOT LESS
         RESET     CWK2 TO 2
*
BUILDANS CLEAR     PROGRAM                BUILD THE NAME
         APPEND    "ANSWER" TO PROGRAM    FORM: ANSWERn
         APPEND    CWK2 TO PROGRAM        WHERE: n IS THE PORT
         RESET     PROGRAM                NUMBER (0 < n < 17)
         TRAP      BADANS IF CFAIL
         CHAIN     PROGRAM
*.....................................................................
. ANSWER PROGRAM COULD NOT BE FOUND
.
BADANS   DISPLAY   *P58:1,*EL,"  The system program":
                   *P58:2,*EL,"  #"",*+,PROGRAM,"#" could not":
                   *P58:3,*EL,"  be found!  Consult":
                   *P58:4,*EL,"  your programmer."
         GOTO      HANG
```

C.1.4 Sample Program Selection MENU

```
. MENU1 - MENU FOR WEEKLY PAYROLL SYSTEM
.
. THIS PROGRAM WAS GENERATED USING THE "MAKEMENU" PROGRAM
.
. COMPILING THIS PROGRAM REQUIRES THAT THE FILES:  "COMMON/TXT",
. "LOGDATA/TXT" AND "LOGIO/TXT" EXIST ON ANY DRIVE WHICH IS ON-
. LINE.  THESE INCLUSION FILES CONTAIN THE INFORMATION COMMON TO
. ALL OF THE SYSTEM PROGRAMS.
.
         INCLUDE   COMMON/TXT
INDEX    FORM      2                    USER SELECTION VARIABLE
TIME     INIT      "hh:mm:ss"           hours:minutes:seconds
         INCLUDE   LOGDATA/TXT
```

```
+.....................................................................
. MAINLINE
.
*.....................................................................
. THIS MENU REQUIRES A SECURITY CLEARANCE OF AT LEAST 2
.
         COMPARE    "2" TO SECURITY
         STOP       IF LESS
         INCLUDE    LOGIO/TXT
*.....................................................................
. DISPLAY THE MENU
.
         DISPLAY    *ES:
                    "MENU FOR WEEKLY PAYROLL SYSTEM":
                    *P51:1,"Today is ",TODAY:
                    *P01:03,"( 1) ":
                    "Enter timecard data":
                    *P01:04,"( 2) ":
                    "Print payroll checks":
                    *P01:05,"( 3) ":
                    "Print check register":
                    *P01:06,"( 4) ":
                    "Enter void checks":
                    *P01:07,"( 5) ":
                    "Print timecard labels":
                    *P01:08,"( 6) ":
                    "Print FICA register":
                    *P01:09,"( 7) ":
                    "Print U/C report":
                    *P01:10,"( 8) ":
                    "Print quarterly FICA report":
                    *P41:03,"( 9) ":
                    "Print W-2's":
                    *P41:04,"(10) ":
                    "Re-organize employee file":
                    *P41:05,"(11) ":
                    "Add new employees":
                    *P41:06,"(12) ":
                    "Change employee master file":
                    *P41:07,"(13) ":
                    "List employee master file":
                    *P41:08,"(14) ":
                    "Print payroll general ledger":
                    *EL
```

```
*.....................................................................
. GET THE PROGRAM'S INDEX
.
GETINDEX KEYIN      *P1:12,*EL,"Selection by number":
                    *P41:12,"Enter (99) to leave this menu.":
                    *P25:12,"__",*P25:12,INDEX
         COMPARE    "1" TO INDEX
         GOTO       GETINDEX IF LESS
         COMPARE    "99" TO INDEX
         STOP       IF EQUAL
         COMPARE    "15" TO INDEX
         GOTO       GETINDEX IF NOT LESS
*.....................................................................
. BRANCH TO THE ROUTINE INDICATED BY THE INDEX
.
         TRAP       BADCHAIN IF CFAIL
         CLOCK      TIME TO TIME
         BRANCH     INDEX OF PAY1:  Enter timecard data
                              PAY2:  Print payroll checks
                              PAY3:  Print check register
                              PAY4:  Enter void checks
                              PAY5:  Print timecard labels
                              PAY6:  Print FICA register
                              PAY7:  Print U/C report
                              PAY8:  Print quarterly FICA report
                              PAY9:  Print W-2's
                              PAY10:  Re-organize employee file
                              PAY11:  Add new employees
                              PAY12:  Change employee master file
                              PAY13:  List employee master file
                              PAY14   Print payroll general ledger
         GOTO       GETINDEX
*.....................................................................
. PROGRAM DOES NOT EXIST.
.
BADCHAIN RETURN
*.....................................................................
. CHAIN INSTRUCTIONS
*.....................................................................
. Enter timecard data
.
PAY1     MOVE       "PROGRAM" TO LOGTYPE
         MOVE       "PAY1    " TO LOGINFO
         CALL       LOGWRITE
         CHAIN      "PAY1"
         WRITAB     LOGFILE,LOGRN;*12,"NO PROGRAM"
         GOTO       GETINDEX
```

```
*.....................................................................
. Print payroll checks
.
PAY2      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY2    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY2"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print check register
.
PAY3      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY3    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY3"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Enter void checks
.
PAY4      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY4    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY4"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print timecard labels
.
PAY5      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY5    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY5"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print FICA register
.
PAY6      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY6    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY6"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
```

```
*.....................................................................
. Print U/C report
.
PAY7      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY7    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY7"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print quarterly FICA report
.
PAY8      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY8    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY8"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print W-2's
.
PAY9      MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY9    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY9"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Re-organize employee file
.
PAY10     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY10   " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY10"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Add new employees
.
PAY11     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY11   " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY11"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
```

```
*.....................................................................
. Change employee master file
.
PAY12     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY12    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY12"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. List employee master file
.
PAY13     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY13    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY13"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
*.....................................................................
. Print payroll general ledger
.
PAY14     MOVE      "PROGRAM" TO LOGTYPE
          MOVE      "PAY14    " TO LOGINFO
          CALL      LOGWRITE
          CHAIN     "PAY14"
          WRITAB    LOGFILE,LOGRN;*12,"NO PROGRAM"
          GOTO      GETINDEX
```

C.1.5 Chain Files for System Generation

The following chain files may be used for system generation and maintenance.

C.1.5.1 Compile the System Programs

```
. MAKEANMA - COMPILE ANSWER AND MASTER PROGRAMS
.
. CHAIN TAGS:  DATE#value#   ==> FORCES LISTING, (#value# USED IN
.                                HEADINGS
.              <number>      ==> FORCES COMPILATION OF MASTER AND
.                                ANSWER PROGRAMS FOR THE PORT
.                                NUMBER SPECIFIED
.              HALF          ==> FORCES COMPILATION OF MASTER AND
.                                ANSWER PROGRAMS FOR PORTS 1-8
.              ALL           ==> FORCES COMPILATION OF MASTER AND
.                                ANSWER PROGRAMS FOR PORTS 1-16
.              SAMPLE        ==> FORCES COMPILATION OF THE SAMPLE
.                                MENUS
.              NEW           ==> FORCES CREATION OF NEW SYSTEM LOG
.                                FILE AND A NEW LIST OF AUTHORIZED
.                                USERS
.
. EXAMPLE:     TO COMPILE THE MASTER AND ANSWER PROGRAMS FOR
.              PORTS 1 THROUGH 4, TO PRODUCE LISTINGS OF ALL
.              PROGRAMS COMPILED, AND TO GENERATE NEW SYSTEMS
.              FILES:  USE THE FOLLOWING DOS COMMAND LINE
.
.              CHAIN MAKEANMA/CHN;1,2,3,4,DATE#ddmmmyy#,NEW
.
.....................................................................
.
// IFS DATE
. I WILL PRODUCE LISTINGS OF THE PROGRAMS
.
// XIF
// IFS SAMPLE
. I WILL COMPILE THE SAMPLE PROGRAMS
// BEGIN
//.
//. COMPILE THE MASTER MENU
//.
// IFS DATE
DBCMP MASMENU;L
DATASHARE MASTER MENU  (MENU SELECTION PROGRAM)              #DATE#
// ELSE
DBCMP MASMENU
// XIF
//.
//. COMPILE A SAMPLE MENU
```

```
//.
// IFS DATE
DBCMP MENU1;L
SAMPLE MENU PROGRAM                                                    #DATE#
// ELSE
DBCMP MENU1
// XIF
// END
// XIF
// IFS 1,HALF,ALL
// BEGIN
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  1
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 1
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 1"
!
//.
//. COMPILE ANSWER1
//.
// IFS DATE
DBCMP ANSWER,ANSWER1;L
DATASHARE ANSWER PROGRAM                                               #DATE#
// ELSE
DBCMP ANSWER,ANSWER1
// XIF
//.
//. COMPILE MASTER1
//.
// IFS DATE
DBCMP MASTER,MASTER1;L
DATASHARE MASTER PROGRAM   (FOR LOGGING DATASHARE ERRORS)  #DATE#
// ELSE
DBCMP MASTER,MASTER1
// XIF
// END
// XIF
// IFS 2,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  2
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 2
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 2"
!
//.
//. COMPILE ANSWER2
```

```
//.
DBCMP ANSWER,ANSWER2
//.
//. COMPILE MASTER2
//.
DBCMP MASTER,MASTER2
// XIF
// IFS 3,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  3
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 3
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 3"
!
//.
//. COMPILE ANSWER3
//.
DBCMP ANSWER,ANSWER3
//.
//. COMPILE MASTER3
//.
DBCMP MASTER,MASTER3
// XIF
// IFS 4,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  4
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 4
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 4"
!
//.
//. COMPILE ANSWER4
//.
DBCMP ANSWER,ANSWER4
//.
//. COMPILE MASTER4
//.
DBCMP MASTER,MASTER4
// XIF
// IFS 5,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  5
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 5
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 5"
```

```
!
//.
//. COMPILE ANSWER5
//.
DBCMP ANSWER,ANSWER5
//.
//. COMPILE MASTER5
//.
DBCMP MASTER,MASTER5
// XIF
// IFS 6,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  6
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 6
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 6"
!
//.
//. COMPILE ANSWER6
//.
DBCMP ANSWER,ANSWER6
//.
//. COMPILE MASTER6
//.
DBCMP MASTER,MASTER6
// XIF
// IFS 7,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  7
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 7
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 7"
!
//.
//. COMPILE ANSWER7
//.
DBCMP ANSWER,ANSWER7
//.
//. COMPILE MASTER7
//.
DBCMP MASTER,MASTER7
// XIF
// IFS 8,HALF,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  8
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 8
```

```
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 8"
!
//.
//. COMPILE ANSWER8
//.
DBCMP ANSWER,ANSWER8
//.
//. COMPILE MASTER8
//.
DBCMP MASTER,MASTER8
// XIF
// IFS 9,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT  9
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 9
//.
BUILD PORTN/TXX;!
PORTN    FORM      " 9"
!
//.
//. COMPILE ANSWER9
//.
DBCMP ANSWER,ANSWER9
//.
//. COMPILE MASTER9
//.
DBCMP MASTER,MASTER9
// XIF
// IFS 10,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 10
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 10
//.
BUILD PORTN/TXX;!
PORTN    FORM      "10"
!
//.
//. COMPILE ANSWER10
//.
DBCMP ANSWER,ANSWER10
//.
//. COMPILE MASTER10
//.
DBCMP MASTER,MASTER10
// XIF
// IFS 11,ALL
```

```
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 11
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 11
//.
BUILD PORTN/TXX;!
PORTN    FORM      "11"
!
//.
//. COMPILE ANSWER11
//.
DBCMP ANSWER,ANSWER11
//.
//. COMPILE MASTER11
//.
DBCMP MASTER,MASTER11
// XIF
// IFS 12,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 12
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 12
//.
BUILD PORTN/TXX;!
PORTN    FORM      "12"
!
//.
//. COMPILE ANSWER12
//.
DBCMP ANSWER,ANSWER12
//.
//. COMPILE MASTER12
//.
DBCMP MASTER,MASTER12
// XIF
// IFS 13,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 13
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 13
//.
BUILD PORTN/TXX;!
PORTN    FORM      "13"
!
//.
//. COMPILE ANSWER13
//.
DBCMP ANSWER,ANSWER13
//.
//. COMPILE MASTER13
//.
```

```
DBCMP MASTER,MASTER13
// XIF
// IFS 14,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 14
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 14
//.
BUILD PORTN/TXX;!
PORTN    FORM      "14"
!
//.
//. COMPILE ANSWER14
//.
DBCMP ANSWER,ANSWER14
//.
//. COMPILE MASTER14
//.
DBCMP MASTER,MASTER14
// XIF
// IFS 15,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 15
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 15
//.
BUILD PORTN/TXX;!
PORTN    FORM      "15"
!
//.
//. COMPILE ANSWER15
//.
DBCMP ANSWER,ANSWER15
//.
//. COMPILE MASTER15
//.
DBCMP MASTER,MASTER15
// XIF
// IFS 16,ALL
. I WILL COMPILE THE MASTER AND ANSWER PROGRAMS FOR PORT 16
//.
//. CREATE THE INCLUSION FILE FOR PORT NUMBER 16
//.
BUILD PORTN/TXX;!
PORTN    FORM      "16"
!
//.
//. COMPILE ANSWER16
//.
DBCMP ANSWER,ANSWER16
```

```
//.
//. COMPILE MASTER16
//.
DBCMP MASTER,MASTER16
// XIF
.
.....................................................................
//.
//. DELETE THE PORT NUMBER INCLUSION FILE
//.
KILL PORTN/TXX
Y
// IFS NEW
. I WILL CREATE NEW SYSTEMS FILES
//.
//. CREATE NEW FILE OF AUTHORIZED USERS
//.
BUILD USERS/DSP;!
   SSN   USER'S NAME      SECURITY
(       )(                 )_( . . .
999999999Anyuser            9
!
//.
//. INDEX THE FILE OF AUTHORIZED USERS ON COLUMNS 1-9
//.
INDEX USERS/DSP,USERS/ISI;1-9
//.
//. CREATE A NEW LOG FILE
//.
CHAIN LOGMAKE/CHN;NEW
// XIF
```

.1.5.2 Re-organize System Log File

```
. LOGMAKE - RE-ORGANIZE DATASHARE SYSTEM 'LOGFILE'
.
. CHAIN TAGS:  NEW         ==> CAUSES A NEW LOG FILE TO BE CREATED
.              REFORMAT    ==> CAUSES THE LOG FILE TO BE REFORMATED
.              LIST#date#  ==> CAUSES THE LOG FILE TO BE LISTED
.                              #date# WILL BE INCLUDED IN THE HEADING
.              SAVE        ==> CAUSES THE LOG FILE TO BE SAVED
.
.....................................................................
.
.
.
.
.
.
// IFS REFORMAT
. I WILL REFORMAT THE LOG FILE
//.
//. REFORMATING THE LOG FILE
//.
REFORMAT LOGFILE;DC
// XIF
// IFS LIST
. I WILL LIST THE LOG FILE
//.
//. LISTING THE LOG FILE
//.
LIST LOGFILE;L
  SYSTEM LOG FILE                                          #LIST#
// XIF
// IFS SAVE
. I WILL SAVE THE LOG FILE BY ADDING IT TO 'MASTERLG/TXT'
//
//. ADDING THE LOG FILE TO THE MASTER LOG FILE
//.
SAPP MASTERLG,LOGFILE,MASTERLG
// XIF
// IFS NEW
. I WILL CREATE A NEW LOG FILE
//.
//. CREATING A NEW LOG FILE
//.
BUILD LOGFILE;!
*000  PORT  LOG TYPE     DATE       TIME     OTHER INFORMATION
```

```
* rn   ()  (        )  (      )  (      )  ( . . .
!
// XIF
.
.....................................................................
.
```

C.2 SYSTEM INCLUSION FILES

The following files are included in the source of all of the system programs to make certain commonly used program segments easier to use.

C.2.1 COMMON User's Data Area

```
*.....................................................................
. COMMON - DEFINE COMMON DATA AREAS
.
PORTN     FORM      *2                  PORT NUMBER
TODAY     DIM       *8                  DATE IN mm/dd/yy FORMAT
SECURITY  FORM      *1                  SECURITY CLEARANCE LEVEL
```

C.2.2 Log File Data Area Definition

```
*.....................................................................
. LOGDATA - UPDATE THE SYSTEM LOG FILE -- INCLUSION FILE #1
.
. THIS FILE CONTAINS THE DATA AREA DEFINITION STATEMENTS THAT ARE
. REQUIRED BY THE LOG FILE I/O ROUTINES
.
* RESTRICTIONS:  -- THIS FILE MAY BE INCLUDED IN A PROGRAM ONLY
.                  ONCE
.               -- THIS FILE MUST BE INCLUDED WITHIN THE
.                  STATEMENTS USED TO DEFINE THE USER'S DATA
.                  AREA
*.....................................................................
. A LOG ENTRY HAS THE FOLLOWING FORMAT:
. POSITIONS   USED FOR:
.   1- 7      RESERVED
.   8- 9      NUMBER OF THE PORT WRITING THE LOG ENTRY
.  10-11      RESERVED
*  12-21      THE LOG ENTRY'S TYPE:
.                  LOG-ON ....... USER SIGN ON
.                  LOG-OFF ...... USER SIGN OFF
.                  BAD ID ....... INVALID ATTEMPT TO SIGN ON
.                  ERROR ........ DATASHARE ERROR
.                  PROGRAM ...... SUCCESSFUL CHAIN TO A PROGRAM
.                  NO PROGRAM ... UNSUCCESSFUL CHAIN TO A PROGRAM
.                  ROLLOUT ...... SUCCESSFUL ROLLOUT
.                  ROLL RET ..... ROLLOUT RETURN
.                  NO ROLLOUT ... UNSUCCESSFUL ROLLOUT
.                  ...
*  22-23      RESERVED
.  24-31      DATE OF LOG ENTRY
.  32-33      RESERVED
.  34-41      TIME OF LOG ENTRY
.  42-43      RESERVED
.  44-64      OTHER INFORMATION:
.                  LOG-ON ....... USER'S NAME
.                  BAD ID ....... INVALID NUMBER ENTERED
.                  ERROR ........ ERROR MESSAGE
.                  PROGRAM ...... NAME OF PROGRAM
.                  NO PROGRAM ... NAME OF PROGRAM
.                  ...
```

```
*.....................................................................
. THE FOLLOWING VARIABLES MUST BE SET TO THEIR APPROPRIATE VALUES
. BEFORE WRITING A LOG ENTRY
.
LOGFILE  FILE                          SYSTEM LOG FILE
LOGTYPE  DIM       10                  TYPE OF LOG TO BE WRITTEN
LOGINFO  DIM       20                  OTHER INFORMATION
*.....................................................................
. THE FOLLOWING VARIABLES MUST BE DEFINED ELSEWHERE AND BE SET TO
. THEIR APPROPRIATE VALUES BEFORE WRITING A LOG ENTRY
.
.PORTN    FORM      2                   PORT NUMBER
.TODAY    INIT      "mm/dd/yy"          month/day/year
.TIME     INIT      "hh:mm:ss"          hours:minutes:seconds
*.....................................................................
. SINCE THE SYSTEM LOG FILE IS COMMON TO ALL PORTS, THE FOLLOWING
. VARIABLES ARE NEEDED TO HANDLE THE COMMON FILE CONSIDERATIONS
.
LOGRN     FORM      3                   RECORD NUMBER OF LOG ENTRY
ZERO      FORM      "0"                 RECORD NUMBER AT RECORD 0
```

C.2.3 Log File Input/Output Routines

```
*.....................................................................
. LOGIO -  UPDATE THE SYSTEM LOG FILE -- INCLUSION FILE #2
.
. THIS FILE CONTAINS:  I.   A ROUTINE THAT OPENS THE SYSTEM LOG
.                           FILE
.                      II.  A SUBROUTINE THAT WRITES A LOG
.                           ENTRY TO THE SYSTEM LOG FILE
.
* RESTRICTIONS:  -- THIS FILE MAY BE INCLUDED IN A PROGRAM ONLY
.                  ONCE
.                -- THIS FILE SHOULD BE INCLUDED IN A PROGRAM AT
.                  THE POINT WHERE THE USER WISHES THE LOG FILE
.                  TO BE OPENED
*.....................................................................
. I.  OPEN THE SYSTEM LOG FILE
.
        TRAP      NOLOG IF IO
        OPEN      LOGFILE,"LOGFILE"
        TRAPCLR   IO
        GOTO      LOGOPEN
*.....................................................................
. LOG FILE IS MISSING
.
NOLOG   DISPLAY   *P54:1,*EL,"  #"LOGFILE/ISI#" is missing!":
                  *P54:2,*EL,"  The port number is ",PORTN
HANG    GOTO      HANG
```

```
*.....................................................................
. II.  WRITE A LOG ENTRY TO THE SYSTEM LOG FILE
.
. PROCEDURE:  1. LOCK OUT ALL OTHER PORTS
.             2. GET THE NUMBER OF LAST USED RECORD (RN)
.             3. PUT AN EOF MARK IN RECORD RN+2 (THIS INSURES
.                THAT THE EOF OF THE LOG FILE IS ALWAYS MARKED)
.             4. PUT RN+1 IN THE LOG FILE AS THE LAST USED RECORD
.             5. ALLOW OTHER PORTS TO EXECUTE
.             6. WRITE THE LOG ENTRY TO RECORD RN+1 (NOTE THAT
.                THIS OVERWRITES THE OLD END-OF-FILE MARK)
.
LOGWRITE PI         5                          1. LOCK OUT
         READ       LOGFILE,ZERO;*2,LOGRN      2. READ RN
         ADD        "2" TO LOGRN
         WEOF       LOGFILE,LOGRN              3. EOF AT RN+2
         SUBTRACT   "1" FROM LOGRN
         WRITAB     LOGFILE,ZERO;*2,LOGRN      4. PUT RN+1
.
. PI GOES TO 0 AT THIS POINT                    5. ALLOW OTHER PORTS
*.....................................................................
. SEE DESCRIPTIONS IN DATA AREA                 6. WRITE LOG ENTRY
.
         WRITE      LOGFILE,LOGRN;"       ",PORTN:
                                  "  ",LOGTYPE:
                                  "  ",TODAY:
                                  "  ",TIME:
                                  "  ",LOGINFO
         RETURN
*.....................................................................
. NOTE:  THE "TRAPCLR PARITY" INSTRUCTION IS USED AS A "NOP"
.        INSTRUCTION
.
LOGOPEN  TRAPCLR    PARITY
```

C.3 SUPLEMENTAL SYSTEM PROGRAMS

Although the following programs are not necessary for using the DATASHARE system defined in this appendix, they should make using and modifying the system much simpler.

C.3.1 Re-organize the List of Authorized Users

```
. NEWUSER - PROGRAM TO UPDATE THE LIST OF AUTHORIZED USERS
.
         INCLUDE   COMMON
*.....................................................................
CFILE    FILE
SEQ      FORM      "-1"
*.....................................................................
USERS    IFILE
USERID   DIM       9
NAME     DIM       20
CLRANCE  FORM      1
*.....................................................................
NWK9     FORM      9
CWK1     DIM       1
REPLY    DIM       1
INDEX    FORM      2                    USER SELECTION VARIABLE
```

```
+.....................................................
. MAINLINE
.
*.....................................................
. THIS MENU REQUIRES A SECURITY CLEARANCE OF AT LEAST 8
.
         COMPARE   "8" TO SECURITY
         STOP      IF LESS
*.....................................................
. PREPARE THE CHAIN FILE
.
         TRAP      NOCHAIN IF IO
         PREPARE   CFILE,"ROLCHAIN"
         TRAPCLR   IO
         WRITE     CFILE,SEQ;*+,"//* TAP THE DISPLAY KEY TO ":
                                "RE-ORGANIZE THE LIST OF ":
                                "AUTHORIZED USERS"
         WRITE     CFILE,SEQ;"//."
         GOTO      OPENUSER
*.....................................................
. CHAIN FILE COULD NOT BE CREATED
.
NOCHAIN  DISPLAY   *ES:
                   *P20:4,"CHAIN FILE COULD NOT BE WRITTEN!":
                   *W,*W
         STOP
*.....................................................
. OPEN THE FILE OF AUTHORIZED USERS
.
OPENUSER TRAP      NOUSER IF IO
         OPEN      USERS,"USERS"
         TRAPCLR   IO
         GOTO      MENU
*.....................................................
. FILE OF AUTHORIZED USERS NOT THERE
.
NOUSER   KEYIN     *ES:
                   *P20:4,"The list of authorized users is":
                   *P20:5,"missing.":
                   *P1:12,"Do you want to create a new list? ":
                   *EL,REPLY
         CMATCH    "Y" TO REPLY
         STOP      IF NOT EQUAL
```

```
*.....................................................................
. CREATE A NEW LIST OF AUTHORIZED USERS
.
         DISPLAY   *ES:
                   *P20:4,"Writing the chain file."
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"//. BUILD THE FILE CONTAINING ":
                              "THE LIST OF AUTHORIZED USERS"
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"BUILD USERS/DSP;!"
         WRITE     CFILE,SEQ;"   SSN   USER'S NAME      ":
                             "SECURITY"
         WRITE     CFILE,SEQ;"(       )(                ":
                             "  )_( . . ."
         WRITE     CFILE,SEQ;"!"
         CALL      CHAINROL
         GOTO      OPENUSER
*.....................................................................
. DISPLAY THE MENU
.
MENU     DISPLAY   *ES:
                   "PROGRAM TO UPDATE THE LIST OF AUTHORIZED USER
                   *P51:1,"Today is ",TODAY:
                   *P01:03,"( 1) ":
                   "Authorize a new user":
                   *P01:04,"( 2) ":
                   "Modify a user's authorization":
                   *P01:05,"( 3) ":
                   "Remove a user from the list":
                   *EL
*.....................................................................
. GET THE PROGRAM'S INDEX
.
GETINDEX KEYIN     *P1:12,*EL,"Selection by number":
                   *P41:12,"Enter (99) to continue.":
                   *P25:12,"__",*P25:12,INDEX
         COMPARE   "1" TO INDEX
         GOTO      GETINDEX IF LESS
         COMPARE   "99" TO INDEX
         GOTO      WRTCHAIN IF EQUAL
         COMPARE   "04" TO INDEX
         GOTO      GETINDEX IF NOT LESS
```

```
*.....................................................................
. BRANCH TO THE ROUTINE INDICATED BY THE INDEX
.
         BRANCH    INDEX OF ADD:  Authorize a new user
                            CHANGE:  Modify a user's authorizatio
                            DELETE   Remove a user from the list
         GOTO      GETINDEX
*.....................................................................
. Authorize a new user
.
*.....................................................................
. DISPLAY THE FORM
.
ADD      DISPLAY   *ES:
                   *P25:6,"________ ____________________ _"
*.....................................................................
. GET THE USER'S ID #
.
GETIDN   CALL      GETID
         CMATCH    " " TO USERID
         GOTO      GETNME IF NOT EOS
*.....................................................................
. ASK IF HE IS DONE WITH THIS ENTRY
.
         KEYIN     *P25:4,*EL:
                   *P1:12,"Are you done? (Y/N) ",*EL,REPLY
         CMATCH    "Y" TO REPLY
         GOTO      MENU IF EQUAL
         GOTO      ADD
*.....................................................................
. GET THE USER'S NAME
.
GETNME   CALL      GETNAME
         GOTO      GETCLR IF NOT EOS
*.....................................................................
. ASK IF DONE WITH THIS ENTRY
.
ASKDONEN KEYIN     *P1:12,"Do you want to re-enter the (I)dent":
                   "ification number or the (N)ame? ",*EL,REPLY
         CMATCH    "N" TO REPLY
         GOTO      GETNME IF EQUAL
         CMATCH    "I" TO REPLY
         GOTO      GETIDN IF EQUAL
         GOTO      ASKDONEN
```

```
*.....................................................................
. GET THE USER'S SECURITY CLEARANCE
.
GETCLR   CALL      GETCLEAR
         COMPARE   "0" TO CLRANCE
         GOTO      WRTNEWU IF NOT EQUAL
*.....................................................................
. ASK IF DONE WITH THIS ENTRY
.
ASKDONEC KEYIN     *P1:12,"Re-enter (I)d number, (N)ame, ":
                   "(C)learance or enter (Z)ero clearance? ":
                   *EL,REPLY
         CMATCH    "I" TO REPLY
         GOTO      GETIDN IF EQUAL
         CMATCH    "N" TO REPLY
         GOTO      GETNME IF EQUAL
         CMATCH    "C" TO REPLY
         GOTO      GETCLR IF EQUAL
         CMATCH    "Z" TO REPLY
         GOTO      ASKDONEC IF NOT EQUAL
*.....................................................................
. ADD THE USER TO THE LIST OF AUTHORIZED USERS
.
WRTNEWU  CALL      INSERT
         GOTO      ADD
*.....................................................................
. Remove a user from the list
.
*.....................................................................
. GET THE USER TO BE DELETED
.
DELETE   CALL      GETUSER
         CMATCH    " " TO USERID
         GOTO      VERIFY IF NOT EOS
*.....................................................................
. ASK IF DONE WITH THIS ENTRY
.
         KEYIN     *P1:12,"Are you done? (Y/N) ",*EL,REPLY
         CMATCH    "Y" TO REPLY
         GOTO      MENU IF EQUAL
         GOTO      DELETE
```

```
*.....................................................................
. MAKE SURE HE WANTS TO DELETE BEFORE REMOVING
.
VERIFY   KEYIN     *P1:12,"Is this the entry to be removed? ":
                   *EL,REPLY
         CMATCH    "Y" TO REPLY
         GOTO      DELETE IF NOT EQUAL
*
         DELETE    USERS,USERID
         GOTO      DELETE
*.....................................................................
. Modify a user's authorization
.
*.....................................................................
. GET THE ENTRY FROM THE LIST TO BE MODIFIED
.
CHANGE   CALL      GETUSER
         CMATCH    " " TO USERID
         GOTO      ASKMOD IF NOT EOS
*.....................................................................
. ASK IF DONE WITH ENTRY
.
         KEYIN     *P1:12,"Are you done? (Y/N) ",*EL,REPLY
         CMATCH    "Y" TO REPLY
         GOTO      MENU IF EQUAL
         GOTO      CHANGE
*.....................................................................
. FIND OUT WHAT HE WANTS TO DO WITH IT
.
ASKMOD   KEYIN     *P1:12,"(D)one, modify (I)d number, ":
                   "modify (N)ame, or ":
                   "modify security (C)learance? ",*EL,REPLY
         CMATCH    "D" TO REPLY
         GOTO      WRTMOD IF EQUAL
         CMATCH    "I" TO REPLY
         GOTO      IDMOD IF EQUAL
         CMATCH    "N" TO REPLY
         GOTO      NAMEMOD IF EQUAL
         CMATCH    "C" TO REPLY
         GOTO      CLRMOD IF EQUAL
         GOTO      ASKMOD
*.....................................................................
. MODIFY THE SECURITY CLEARANCE
.
CLRMOD   CALL      GETCLEAR
         COMPARE   "0" TO CLRANCE
         GOTO      ASKMOD IF NOT EQUAL
```

```
*.....................................................................
. ASK IF DONE WITH ENTRY
.
ASKDONEZ KEYIN      *P1:12,"(D)one or enter (Z)ero security ":
                    "clearance? ",*EL,REPLY
         CMATCH     "D" TO REPLY
         GOTO       WRTMOD IF EQUAL
         CMATCH     "Z" TO REPLY
         GOTO       ASKDONEZ IF NOT EQUAL
         GOTO       ASKMOD
*.....................................................................
. MODIFY THE NAME
.
NAMEMOD  CALL       GETNAME
         GOTO       ASKMOD IF NOT EOS
*.....................................................................
. ASK IF DONE WITH ENTRY
.
         KEYIN      *P1:12,"Are you done? (Y/N) ",*EL,REPLY
         CMATCH     "Y" TO REPLY
         GOTO       WRTMOD IF EQUAL
         GOTO       ASKMOD
*.....................................................................
. MODIFY THE IDENTIFICATION NUMBER
.
*.....................................................................
. DELETE THE OLD USER ID
.
IDMOD    DELETE     USERS,USERID
*.....................................................................
. GET THE NEW ID NUMBER
.
NEWID    CALL       GETID
         CMATCH     " " TO USERID
         GOTO       NEWID IF EOS
*.....................................................................
. INSERT THE NEW USER INTO THE LIST OF AUTHORIZED USERS
.
         CALL       INSERT
         GOTO       CHANGE
*.....................................................................
. UPDATE THE ENTRY
.
WRTMOD   UPDATE     USERS;USERID,NAME,CLRANCE
         GOTO       CHANGE
```

```
*.....................................................................
. WRITE THE CHAIN FILE
.
WRTCHAIN DISPLAY    *ES,*P25:4,"Writing the CHAIN file."
         CALL       CHAINROL
         STOP
```

```
+.....................................................................
. GET THE USER'S ID NUMBER
.
GETID     CLEAR     USERID
          KEYIN     *P25:4,"To exit, tap the ENTER key":
                    *P25:6,"_________",*P1:12:
                    "Enter the user's identification number.",*EL:
                    *P25:6,NWK9:
                    *P25:4,*EL
          COMPARE   "0" TO NWK9
          RETURN    IF EQUAL
          COMPARE   "100000000" TO NWK9
          GOTO      GETID IF LESS
          MOVE      NWK9 TO USERID
          RETURN
*.....................................................................
. GET THE USER'S SECURITY CLEARANCE
.
GETCLEAR  KEYIN     *P25:4,"To exit, tap the ENTER key":
                    *P56:6,"_",*P1:12:
                    "Enter the user's security clearance.",*EL:
                    *P56:6,CLRANCE:
                    *P25:4,*EL
          DISPLAY   *P56:6,CLRANCE
          MOVE      CLRANCE TO CWK1
          CMATCH    "-" TO CWK1
          GOTO      GETCLEAR IF EQUAL
          RETURN
*.....................................................................
. GET THE USER'S NAME
.
GETNAME   KEYIN     *P25:4,"To exit, tap the ENTER key":
                    *P35:6,"____________________",*P1:12:
                    "Enter the user's name.",*EL:
                    *P35:6,*IT,NAME,*IN:
                    *P25:4,*EL
          DISPLAY   *P35:6,NAME
          CMATCH    " " TO NAME
          GOTO      GETNAME IF EQUAL
          RETURN
*.....................................................................
. GET AND DISPLAY AN ENTRY FROM THE LIST OF AUTHORIZED USERS
.
```

```
*.....................................................
. GET THE USER'S ID #
.
GETUSER  DISPLAY   *ES
         CALL      GETID
         CMATCH    " " TO USERID
         RETURN    IF EOS
*.....................................................
. SEE IF THE USER IS ACTUALLY ON THE LIST
.
         READ      USERS,USERID;USERID,NAME,CLRANCE
         GOTO      SHOWUSER IF NOT OVER
*.....................................................
. USER NOT FOUND
.
         BEEP
         DISPLAY   *P25:4,"That user could not be found",*EL:
                   *W,*W
         GOTO      GETUSER
*.....................................................
. PUT THE ENTRY ONTO THE SCREEN
.
SHOWUSER DISPLAY   *ES,*P25:4,"That user is:":
                   *P25:6,USERID," ",NAME," ",CLRANCE
         RETURN
*.....................................................
. INSERT A NEW USER INTO THE LIST OF AUTHORIZED USERS
.
INSERT   TRAP      NOWRITE IF IO
         WRITE     USERS,USERID;NWK9,NAME,CLRANCE
         TRAPCLR   IO
         RETURN
*.....................................................
. USER ID IS ALREADY IN USE
.
NOWRITE  BEEP
         DISPLAY   *P1:12,*EL:
                   *P25:4,"That user id. is already in use!",*EL:
                   *W,*W,*W
         RETURN
*.....................................................
  WRITE THE CHAIN FILE
```

```
                              "//."
         WRITE     CFILE,SEQ;"//. REFORMAT THE LIST OF ":
                                 "AUTHORIZED USERS"
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"REFORMAT USERS/DSP;R"
*.....................................................
. WRITE THE INDEX LINES
.
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"//. INDEX THE LIST OF ":
                                 "AUTHORIZED USERS"
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"INDEX USERS/DSP;1-9"
*.....................................................
. WRITE ROLLOUT RETURN LINES
.
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"//. RETURN TO DATASHARE"
         WRITE     CFILE,SEQ;"//."
         WRITE     CFILE,SEQ;"DSBACKTD"
*.....................................................
. WRITE EOF'S TO THE FILES
.
         WEOF      CFILE,SEQ
*.....................................................
. ROLLOUT TO THE CHAIN FILE
.
         DISPLAY   *ES,*P25:4,"Rolling out to reorganize the":
                   *P25:5,"file of authorized users."
         TRAP      NOROLL IF CFAIL
         ROLLOUT   "CHAIN ROLCHAIN"
         TRAPCLR   CFAIL
         RETURN
*.....................................................
. ROLLOUT NOT POSSIBLE
.
NOROLL   KEYIN     *ES:
                   *P20:4,"The chain file has been written, but":
                   *P20:5,"the rollout to it failed.  Use the":
                   *P20:6,"following DOS command line to update":
                   *P20:7,"the list of authorized users:":
                   *P20:8,"#"CHAIN ROLCHAIN#"",REPLY
         STOP
```

C.3.2 Program to Generate New Menus

```
. MAKEMENU - MENU GENERATION PROGRAM
.
         INCLUDE   COMMON
*.....................................................................
NAME     DIM       8                     NAME OF MENU / NAME OF
.                                        PROGRAM FOR CHAIN INST.
TITLE    DIM       50                    TITLE TO BE DISPLAYED
REPLY    DIM       1                     REPLY TO QUESTIONS
*.....................................................................
BRANCH   INIT      "BRANCH    INDEX OF "    SEE BELOW
.
. WHEN WRITING THE BRANCH INSTRUCTION, THE STRING ABOVE MUST BE
. WRITTEN PRECEDING THE FIRST PROGRAM NAME ONLY.  A STRING OF
. BLANKS MUST BE WRITTEN PRECEDING ALL OTHER PROGRAM NAMES.  THIS
. IS HANDLED USING THE VARIABLE "BRANCH".  THE VARIABLE "BRANCH"
. IS WRITTEN PRECEDING THE PROGRAM NAME FOR ALL LINES OF THE
. BRANCH INSTRUCTION.  THE FIRST TIME "BRANCH" IS WRITTEN IT
. CONTAINS THE STRING GIVEN ABOVE.  AFTER WRITING THE VARIABLE
. "BRANCH" A STRING OF ALL BLANKS IS MOVED INTO IT CAUSING ALL
. SUBSEQUENT WRITES USING "BRANCH" TO WRITE BLANKS PRECEDING THE
. PROGRAM NAME.
.
*.....................................................................
. THESE VARIABLE ARE USED BY THIS PROGRAM TO POSITION TO THE
. PROPER POSITION ON THE SCREEN AS WELL AS BEING USED TO WRITE
. THE *P<h>:<v> CONTROLS FOR DISPLAYING THE MENU.
.
INDEX    FORM      2                     NUMBER INDICATING WHICH
.                                        PROGRAM
HPOS     FORM      2                     HORIZONTAL POSITION
VPOS     FORM      2                     VERTICAL POSTION
*.....................................................................
. UTILITY WORK AREAS
. C => CHARACTER STRING VARIABLE
. N => NUMERIC STRING VARIABLE
. NUMBER IN LABEL INDICATES THE LENGTH OF THE WORK AREA
.
CWK9     DIM       9
CWK34    DIM       34
CWK65    DIM       65
NWK1     FORM      1
```

```
*.....................................................................
OUTFILE  FILE                              ON COMPLETION CONTAINS THE
.                                          MENU THAT WAS BUILT
WKFILE1  FILE                              WORK FILE USED TO STORE
.                                          BRANCH INSTRUCTION
WKFILE2  FILE                              WORK FILE USED TO STORE
.                                          THE CHAIN INSTRUCTION
.                                          SECTION OF THE MENU
SEQ      FORM      "-1"                    USED FOR SEQUENTIAL I/O
REWIND   FORM      "0"                     USED TO REWIND FILES
```

```
+.....................................................................
. MAINLINE
.
         COMPARE   "8" TO SECURITY     REQUIRE A SECURITY CLEAR-
         STOP      IF LESS             ANCE OF AT LEAST 8
*.....................................................................
. GET THE NAME OF THE MENU
.
BADMENU  KEYIN     *ES,"What is the name of the menu? ",NAME
         CMATCH    " " TO NAME
         GOTO      BADMENU IF EOS
         GOTO      BADMENU IF EQUAL
*.....................................................................
. PREPARE THE OUTPUT FILE
.
         TRAP      PREPOUT IF IO
         OPEN      OUTFILE,NAME
         TRAPCLR   IO
*.....................................................................
. FILE ALREADY EXISTS
.
         KEYIN     "That menu already exists!":
                   *N,"Do you want to overwrite it? (Y/N) ",REPLY
         CMATCH    "Y",REPLY
         GOTO      DATAREA IF EQUAL
         STOP
*.....................................................................
. PREPARE THE OUTPUT FILE
.
PREPOUT  PREPARE   OUTFILE,NAME
*.....................................................................
. DATA AREA GENERATION
.
DATAREA  KEYIN     "What is the menu's title? ",TITLE
*
         CLEAR     CWK65               BUILD THE FIRST COMMENT
         APPEND    NAME TO CWK65
         APPEND    " - " TO CWK65
         APPEND    TITLE TO CWK65
         RESET     CWK65
```

```
*.....................................................
. WRITE THE OPENING COMMENTS
.
        WRITE     OUTFILE,SEQ;*+,". ",CWK65
        WRITE     OUTFILE,SEQ;"."
        WRITE     OUTFILE,SEQ;". THIS PROGRAM WAS GENERATED ":
                              "USING THE #"MAKEMENU#" PROGRAM"
        WRITE     OUTFILE,SEQ;"."
        WRITE     OUTFILE,SEQ;". COMPILING THIS PROGRAM ":
                  "REQUIRES THAT THE FILES:  #"COMMON/TXT#","
        WRITE     OUTFILE,SEQ;". #"LOGDATA/TXT#" AND ":
                  "#"LOGIO/TXT#" EXIST ON ANY DRIVE ":
                  "WHICH IS ON-"
        WRITE     OUTFILE,SEQ;". LINE.  THESE INCLUSION FILES ":
                  "CONTAIN THE INFORMATION COMMON TO"
        WRITE     OUTFILE,SEQ;". ALL OF THE SYSTEM PROGRAMS."
        WRITE     OUTFILE,SEQ;"."
*.....................................................
. WRITE THE USER'S DATA AREA
.
        WRITE     OUTFILE,SEQ;"          INCLUDE   COMMON/TXT"
        WRITE     OUTFILE,SEQ;"INDEX     FORM      2":
                              "                    ":
                              "USER SELECTION VARIABLE"
        WRITE     OUTFILE,SEQ;"TIME      INIT":
                              "       #"hh:mm:ss#"":
                              "            ":
                              "hours:minutes:seconds"
        WRITE     OUTFILE,SEQ;"          INCLUDE   LOGDATA/TXT"
*.....................................................
. START WRITING THE MAINLINE
.
        WRITE     OUTFILE,SEQ;"+.....................":
                              ".....................":
                              "............."
        WRITE     OUTFILE,SEQ;". MAINLINE"
        WRITE     OUTFILE,SEQ;"."
```

```
*.....................................................
. SET UP SECURITY CHECK
.
         KEYIN     "What security clearance should be required":
                   " to execute this menu? (1-9) ",NWK1
         WRITE     OUTFILE,SEQ;"*.....................":
                               ".....................":
                               "............."
         WRITE     OUTFILE,SEQ;". THIS MENU REQUIRES A ":
                               "SECURITY CLEARANCE OF AT ":
                               "LEAST ",NWK1
         WRITE     OUTFILE,SEQ;"."
         WRITE     OUTFILE,SEQ;"         COMPARE":
                               "   #"",NWK1,"#" TO SECURITY"
         WRITE     OUTFILE,SEQ;"         STOP      IF LESS"
         WRITE     OUTFILE,SEQ;"         INCLUDE   LOGIO/TXT"
*.....................................................
. WRITE THE INITIAL PART OF THE MENU DISPLAY INSTRUCTION
.
         WRITE     OUTFILE,SEQ;"*.....................":
                               ".....................":
                               "............."
         WRITE     OUTFILE,SEQ;". DISPLAY THE MENU"
         WRITE     OUTFILE,SEQ;"."
         WRITE     OUTFILE,SEQ;"         DISPLAY   *ES:"
*
         CLEAR     CWK65
         APPEND    TITLE TO CWK65
         APPEND    "#":" TO CWK65
         RESET     CWK65
         WRITE     OUTFILE,SEQ;"                   #"",CWK65
         WRITE     OUTFILE,SEQ;"                   ":
                               "*P51:1,#"Today is #",TODAY:"
*.....................................................
. PREPARE THE WORK FILES
.
         TRAP      NOWORK IF IO
         PREPARE   WKFILE1,"WKFILE1"   USE FOR "BRANCH" INSTRUCT.
         PREPARE   WKFILE2,"WKFILE2"   USE FOR "CHAIN" SECTION
         GOTO      GETMENU
*.....................................................
. WORK FILES COULD NOT BE CREATED
.
NOWORK   DISPLAY   "Work file could not be created!"
         STOP
```

```
*.....................................................................
. INITIALIZE FOR GETTING THE MENU
.
GETMENU  DISPLAY   *ES,*+,TITLE,*P51:1,"Today is ",TODAY
         MOVE      "1" TO HPOS
         MOVE      "3" TO VPOS
         MOVE      "1" TO INDEX
         GOTO      GETITEM
*.....................................................................
. THE LOOP FOR GETTING THE MENU BEGINS HERE.
. THE FOLLOWING ORGANIZATION IS USED FOR THE LOOP SO THAT THE
. LAST LINE OF THE "BRANCH" INSTRUCTION WILL NOT BE WRITTEN UNTIL
. AFTER LEAVING THIS LOOP:
.
. 1. WRITE LINE OF "BRANCH" INST.
. 2. GET NEXT ITEM FROM KEYBOARD     <-- THE LOOP IS ENTERED HERE
. 3. WRITE "LINE" OF CHAIN SECTION
. 4. IF NOT LAST ITEM, GO TO 1.
. 5. WRITE LAST LINE OF "BRANCH"
.
*.....................................................................
. 1. WRITE A LINE OF THE BRANCH INSTRUCTION
.
. WRITE THE BRANCH INSTRUCTION TO A WORK FILE TO BE COPIED TO THE
. OUTPUT FILE AT A LATER TIME
.
WRITEBR  CLEAR     CWK65
         APPEND    "          " TO CWK65    NULL LABEL FIELD.
         APPEND    BRANCH TO CWK65          EXCEPT FOR 1ST TIME
.                                           NULL OPERATION FIELD.
         APPEND    NAME TO CWK65            PROGRAM NAME NEXT
         APPEND    ":  " TO CWK65           ATTACH CONTINUATION ":"
         APPEND    CWK34 TO CWK65           USE PROGRAM DESCRIPTION
.                                           AS COMMENT FIELD
*.....................................................................
. WRITE THE LINE OF THE BRANCH INSTRUCTION
.
. MAKE SURE THAT THE NEXT LINE OF THE BRANCH INSTRUCTION WILL
. HAVE A NULL OPERATION FIELD
.
         RESET     CWK65
         WRITE     WKFILE1,SEQ;*+,CWK65
         MOVE      "                    " TO BRANCH
*.....................................................................
. 2. GET AN ITEM FROM THE KEYBOARD
.
```

```
*.....................................................
. GET THE PROGRAM NAME
.
GETITEM  KEYIN     *P1:12,*EL,"Enter the name of a program to ":
                   "which this menu will CHAIN: ",NAME
         CMATCH    " " TO NAME
         GOTO      GETITEM IF EOS
         GOTO      GETITEM IF EQUAL
*.....................................................
. GET THE PROGRAM DESCRIPTION
.
. NOTE THAT; THE VERTICAL AND HORIZONTAL POSITIONS USED TO GET
. THE DESCRIPTION ARE THE SAME AS THE POSITIONS PUT INTO THE
. MENU PROGRAM WHILE DISPLAYING THE MENU
.
         DISPLAY   *PHPOS:VPOS,"(",INDEX,") ":             PROMPT
                   "______Describe_this_program._____"
         DISPLAY   *PHPOS:VPOS,"(",INDEX,") ";            RE-POSITION
         KEYIN     *IT,CWK34,*IN,*EL                       DATA ENTRY
*.....................................................
. WRITE THE DISPLAY POSITIONING FOR THIS ITEM
.
         WRITE     OUTFILE,SEQ;"                    ":
                               "*P",*ZF,HPOS,":",*ZF,VPOS:
                               ",#"(",INDEX,") #":"
*.....................................................
. CAUSE DISPLAY OF THE PROGRAM DESCRIPTION
.
         CLEAR     CWK65
         APPEND    CWK34 TO CWK65
         APPEND    "#":" TO CWK65
         RESET     CWK65
         WRITE     OUTFILE,SEQ;"                      #"",CWK65
*.....................................................
. 3. WRITE A "LINE" OF THE CHAIN INSTRUCTIONS
.
. WHERE: "LINE" INCLUDES ALL OF THE INSTRUCTIONS NEEDED BEFORE
.        AND AFTER THE ACTUAL CHAIN INSTRUCTION
.
. THESE INSTRUCTIONS ARE WRITTEN TO A WORK FILE TO BE COPIED TO
. THE OUTPUT FILE AT A LATER TIME
```

```
*.....................................................................
. PUT A DOUBLE QUOTE AFTER THE PROGRAM NAME AND LEAVE IN CWK9
.
         CLEAR     CWK9
         APPEND    NAME TO CWK9
         APPEND    "#"" TO CWK9
         RESET     CWK9
*.....................................................................
. WRITE COMMENTS TO PRECEDE INSTRUCTIONS THAT CAUSE CHAIN TO THE
. PROGRAM
.
         WRITE     WKFILE2,SEQ;*+,"*.....................":
                               ".....................":
                               "............."
         WRITE     WKFILE2,SEQ;". ",CWK34
         WRITE     WKFILE2,SEQ;"."
*.....................................................................
. WRITE THE INSTRUCTIONS
.
         WRITE     WKFILE2,SEQ;NAME," MOVE      #"PROGRAM#" ":
                               "TO LOGTYPE"
         WRITE     WKFILE2,SEQ;"          MOVE      #"",NAME:
                               "#" TO LOGINFO"
         WRITE     WKFILE2,SEQ;"          CALL      LOGWRITE"
         WRITE     WKFILE2,SEQ;"          CHAIN     #"",CWK9
         WRITE     WKFILE2,SEQ;"          WRITAB    LOGFILE,":
                               "LOGRN;*12,#"NO PROGRAM#""
         WRITE     WKFILE2,SEQ;"          GOTO      GETINDEX"
*.....................................................................
. 4.  IF THE LAST ITEM, GO TO 5.
.    IF NOT THE LAST ITEM, GOT TO 1.
.
         COMPARE   "16" TO INDEX       NO MORE THAN 16 ITEMS
         GOTO      ENDLOOP IF NOT LESS
*
BADANS   KEYIN     *P1:12,*EL,"Are there any more programs to ":
                   "be included? ",*+,REPLY
         CMATCH    "N" TO REPLY
         GOTO      ENDLOOP IF EQUAL    REQUIRE YES OR NO ANSWER
         CMATCH    "Y" TO REPLY
         GOTO      BADANS IF NOT EQUAL
```

```
*.....................................................................
. BUMP THE INDEX, VERTICAL POSITION AND THE HORIZONTAL POSITION
. BEFORE GOING TO 1.
.
         ADD        "1" TO INDEX
         ADD        "1" TO VPOS
         COMPARE    "9" TO INDEX
         GOTO       WRITEBR IF NOT EQUAL
         MOVE       "3" TO VPOS
         MOVE       "41" TO HPOS
         GOTO       WRITEBR
*.....................................................................
. 5. WRITE THE LAST LINE OF THE BRANCH INSTRUCTION
. (LAST LINE OF BRANCH INSTRUCTION CANNOT HAVE A COLON FOLLOWING)
.
ENDLOOP  CLEAR      CWK65
         APPEND     "          " TO CWK65
         APPEND     BRANCH TO CWK65
         APPEND     NAME TO CWK65
         APPEND     "   " TO CWK65
         APPEND     CWK34 TO CWK65
         RESET      CWK65
         WRITE      WKFILE1,SEQ;CWK65
*.....................................................................
. WRITE END OF FILES TO THE WORK FILES
.
         WEOF       WKFILE1,SEQ
         WEOF       WKFILE2,SEQ
*.....................................................................
. WRITE THE LAST LINE OF THE MENU DISPLAY INSTRUCTION
.
         WRITE      OUTFILE,SEQ;"                    *EL"
*.....................................................................
. WRITE THE ROUTINE TO PROMPT AND KEYIN THE INDEX
.
         DISPLAY    *ES,"Writing KEYIN routine."
         WRITE      OUTFILE,SEQ;"*.....................":
                                ".....................":
                                "............."
         WRITE      OUTFILE,SEQ;". GET THE PROGRAM'S INDEX"
         WRITE      OUTFILE,SEQ;"."
```

```
*.....................................................................
. WRITE THE INSTRUCTIONS THAT DISPLAY THE PROMPTING MESSAGE
.
         WRITE     OUTFILE,SEQ;"GETINDEX KEYIN      *P1:12,":
                               "*EL,#"Selection by number#":"
         WRITE     OUTFILE,SEQ;"                    *P41:12,":
                               "#"Enter (99) to leave this ":
                               "menu.#":"
         WRITE     OUTFILE,SEQ;"                    *P25:12,":
                               "#"__#",*P25:12,INDEX"
*.....................................................................
. WRITE THE INSTRUCTIONS THAT DO THE RANGE CHECK ON THE INDEX
.
         WRITE     OUTFILE,SEQ;"          COMPARE   #"1#"":
                               " TO INDEX"
         WRITE     OUTFILE,SEQ;"          GOTO      GETINDEX ":
                               "IF LESS"
         WRITE     OUTFILE,SEQ;"          COMPARE   #"99#" ":
                               "TO INDEX"
         WRITE     OUTFILE,SEQ;"          STOP      IF EQUAL"
         ADD       "1" TO INDEX
         WRITE     OUTFILE,SEQ;"          COMPARE   #"":
                               *ZF,INDEX,"#" TO INDEX"
         WRITE     OUTFILE,SEQ;"          GOTO      GETINDEX ":
                               "IF NOT LESS"
*.....................................................................
. COPY THE BRANCH INSTRUCTION FROM THE WORK FILE
.
         DISPLAY   *ES,"Writing the BRANCH instruction."
         WRITE     OUTFILE,SEQ;"*.....................":
                               ".....................":
                               "............."
         WRITE     OUTFILE,SEQ;". BRANCH TO THE ROUTINE ":
                               "INDICATED BY THE INDEX"
         WRITE     OUTFILE,SEQ;"."
         WRITE     OUTFILE,SEQ;"          TRAP      BADCHAIN ":
                               "IF CFAIL"
         WRITE     OUTFILE,SEQ;"          CLOCK     TIME ":
                               "TO TIME"
         READ      WKFILE1,REWIND;;
         GOTO      BEGINBRL
*.....................................................................
. GET THE ACTUAL BRANCH STATEMENT FROM WORK FILE 1
.
BRLOOP   WRITE     OUTFILE,SEQ;CWK65
BEGINBRL READ      WKFILE1,SEQ;CWK65
         GOTO      BRLOOP IF NOT OVER
```

```
*
         WRITE     OUTFILE,SEQ;"         GOTO      GETINDEX"
*
         WRITE     OUTFILE,SEQ;"*.......................":
                               ".......................":
                               "............."
         WRITE     OUTFILE,SEQ;". PROGRAM DOES NOT EXIST."
         WRITE     OUTFILE,SEQ;"."
         WRITE     OUTFILE,SEQ;"BADCHAIN RETURN"
*.....................................................
. COPY THE CHAIN INSTRUCTION SECTION FROM THE WORK FILE
.
         DISPLAY   *ES,"Writing the CHAIN instructions."
         WRITE     OUTFILE,SEQ;"*.......................":
                               ".......................":
                               "............."
         WRITE     OUTFILE,SEQ;". CHAIN INSTRUCTIONS"
         READ      WKFILE2,REWIND;;
         GOTO      BEGINCHL
*.....................................................
. GET THE ACTUAL CHAIN INSTRUCTIONS FROM WORK FILE 2
.
CHLOOP   WRITE     OUTFILE,SEQ;CWK65
BEGINCHL READ      WKFILE2,SEQ;CWK65
         GOTO      CHLOOP IF NOT OVER
*.....................................................
. WRITE AN END OF FILE MARK TO THE OUTPUT FILE
. KILL OFF THE WORK FILES
.
         WEOF      OUTFILE,SEQ
*
         PREPARE   WKFILE1,"WKFILE1"
         CLOSE     WKFILE1
*
         PREPARE   WKFILE2,"WKFILE2"
         CLOSE     WKFILE2
```

APPENDIX D. COMMON FILE ACCESS CONSIDERATIONS

Since DATASHARE is capable of executing more than one program concurrently, more than one program at a time can try to access a single file. There is no problem if these accesses are not modifying the contents of the file or if they are dealing with different records in the file. If this is the case, one program will have no idea that another is accessing the same file. However, if a certain record in the file is to be modified by more than one program at a time, a lockout mechanism is needed to allow one program to finish its modification before the other can start. The Prevent Interruptions instruction is provided for this purpose. The PI instruction can solve many common file update conflicts directly as shown in the example in Section 6.12. However, there are cases where several files may have to be read and then a decision made by the operator before the modification can take place. In this case, the part of the record that is going to be modified can be read first and saved. Then the other reads and operator decisions are made and a new value made ready for the modification write. However, before the modification is actually made, interruptions are prevented while the value currently in the record is read again and compared to the value read the first time. If the value has not changed, then the modification is made before interrupts are allowed again. If the value has changed, a new modification value is computed based upon the new value in the location to be updated (this may require another operator decision) and the cycle is repeated. It is assumed that the conflict rate over a given record in a file will be low and the number of times an operator will be asked to repeat a decision will be small. See the example below for an illustration.

Another potential problem regarding common files that are being accessed by more than one port simultaneously exists. This problem is encountered when more than one port is updating a common file. For example, suppose that port A was adding records to the same file as port B and that both ports had new file space allocated. If port A perfomed a CLOSE instruction on the common file, space deallocation would occur on the file and some of the information that port B had written may be lost. A solution to this space deallocation problem is to avoid the use of the CLOSE instruction on the common files.

```
.
. FILE ACCESS LOCKOUT EXAMPLE
.
```

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

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

APPENDIX E. COMPILER ERROR CODES

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

00001 The first operand of a CMATCH or CMOVE instruction was not an octal number, a quoted character, or a string variable.

00002 The second operand of a CMATCH instruction was not an octal number, a quoted character, or a string variable.

00003 The second operand of a MATCH or APPEND instruction was not a string variable.

00004 The first operand of a MATCH or APPEND instruction was not a string variable, numeric variable or a literal.

00005 The first operand of a RESET or ACALL instruction was not a string variable.

00006 The second operand of a RESET instruction was followed by a character that was not a space, implying that there were other operands following the second operand. RESET may have only one or two operands.

00007 The first operand of a BUMP instruction was not a string variable.

00010 The second operand of a BUMP instruction was not terminated by a space, or had an absolute value of greater than 127.

00011 The operand of a CHAIN or ROLLOUT instruction was not a string variable or a literal.

00012 The first operand of a STORE instruction was not a string variable or numeric variable or literal. The first operand of a LOAD instruction was not a string variable or numeric variable.

00013 The second operand of a STORE or LOAD instruction was not a numeric variable.

00014 The second operand of a STORE or LOAD instruction was not followed by either a space or a comma.

00015 One of the third thru Nth operands of a STORE or LOAD instruction was not the same data type as the first operand. If the first operand is a string or numeric variable, then all operands after and including the third operand must be a string or numeric variable, respectively.

00016 The second operand of a MOVE instruction was not a string variable or a numeric variable.

00020 The first operand of a MOVE instruction was not a string variable or a numeric variable or a literal.

00021 The second operand of a COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not a numeric variable.

00022 The second operand of a CMATCH, CMOVE, MATCH, APPEND, CHAIN, ROLLOUT, COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not followed by a space (indicating no more operands follow).

00023 The first operand of a COMPARE, ADD, SUBTRACT, MULTIPLY, or DIVIDE instruction was not a numeric variable or a literal.

00024 The first operand of an instruction which may be followed by a comma or a preposition was not immediately followed by a comma or a space. If a comma follows the operand a preposition is not looked for. If a space does follow the operand then a preposition must be there.

00025 The first operand of a GOTO, CALL, or TRAP instruction was not followed by a space.

00026 The first operand of a TRAP instruction was not followed by " IF ".

00027 The conditional operand ([NOT] EOS, EQUAL, ZERO, etc.) of a GOTO, CALL, or TRAP instruction was not followed by a space.

00030 The conditional operand of a GOTO or CALL instruction was not [NOT] EOS, EQUAL, ZERO, LESS, or OVER; or the conditional operand of a TRAP instruction was not PARITY,

RANGE, FORMAT, CFAIL, or IO.

00031 The first operand of the TRAPCLR instruction was not followed by a space.

00032 The first operand of the TRAPCLR instruction was not PARITY, RANGE, FORMAT, CFAIL, or IO.

00033 An operand in a CONSOLE, KEYIN, or DISPLAY instruction was not a string variable or a numeric variable. It was an EQU, FILE, RFILE, IFILE, or RIFILE variable.

00034 A control code (letter or letters following an asterisk) in a CONSOLE, KEYIN, or DISPLAY instruction was not *C, *L, *N, *T, *R, *P, *EL, *EF, *ES, *W, *EON, *EOFF, *JL, *JR, *ZF, *DE, *IT, OR *IN.

00035 A variable <N> in the *P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was not a number (did not have a first character of 0-9) nor a numeric variable.

00036 A variable <N> in the *P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was a numeric literal with a value for the first (horizontal position) <N> that was not 1 =< <N> =< 80, or with a value for the second (vertical position) <N> that was not 1 =< <N> =< 24.

00037 A literal in a CONSOLE, KEYIN, or DISPLAY instruction was not followed by a comma, space, semicolon, or full colon.

00040 The last character in the operand string of a CONSOLE, KEYIN, DISPLAY, PRINT, RPRINT, READ, WRITE, or WRITAB instruction was not a space, colon, or semicolon.

00041 The end-of-line was encountered before an operand string terminator was encountered for a CONSOLE, KEYIN, DISPLAY, PRINT, RPRINT, READ, WRITE, WRITAB, WEOF, READKS, UPDATE, OPEN, PREPARE, INSERT, or DELETE instruction, or

The character following the first <N> in the *P<N>:<N> control code of a CONSOLE, KEYIN, or DISPLAY instruction was not a colon, or

A quoted string or octal number was specified in the operand string of a READ instruction.

00042 An EQUATE, FILE, RFILE, IFILE, or RIFILE name was specified in the operand list of a PRINT or RPRINT instruction.

00043 A character following an asterisk indicating a control code in a PRINT or RPRINT instruction was not +, -, L, F, C, N, or a number 0-9.

00044 The first operand of a READ, WRITE, WRITAB, or WEOF instruction was not a FILE, RFILE, IFILE, or RIFILE name.

00045 The character following the first operand of a READ, WRITE, WRITAB, or WEOF instruction was not a comma.

00046 The second operand of a READ, WRITE, WRITAB, or WEOF instruction having an IFILE or RIFILE name as the first operand was not a string variable name nor a numeric variable name.

00047 The second operand of a READ, WRITE, WRITAB, or WEOF instruction having a FILE or RFILE name as the first operand was not a numeric variable.

00050 The character following the first operand of a READKS instruction or the second operand of a READ instruction was not a semicolon.

00051 The character following the first opernad of an UPDATE instruction or the second operand of a WRITE instruction was not a space or semicolon.

00052 An operand in the operand string of a READ or READKS instruction was not a tab (*<number> or *<nvar> or *<EQUname>) nor numeric variable nor string variable, or

An oprand in the operand string of a WRITE or UPDATE instruction was not a space compression control (*+ or *-) or a quoted string or numeric variable or string variable, or

An operand in the operand string of a WRITAB or UPDATE instruction was not a tab (*<number> or *<EQUname>) or space compression control (*+ or *-) or quoted string or numeric variable or string varible.

00053 A tab operand (*<number> or *<EQUname> or *<nvar>) was used in a READ instruction that had an IFILE or RIFILE name as operand one and an NVAR name as operand two.

00054 The character following the * control-indicator character in a WRITE instruction was not a + or -. The compiler will recognize only the *+ or *- control for the WRITE

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

00055 The operand following an * control-indicator character was a quoted item. Numeric literals may be used but they may not be enclosed in double-quote " symbols. Numeric literals, numeric variable names, or equated names may be used to specify tab values in KEYIN, DISPLAY, CONSOLE, READ, WRITAB, READKS, or UPDATE instructions.

00056 The operand following an * control-indicator character was not an unquoted numeric literal, a numeric variable name, or an equated name.

00057 The first operand of a READKS or UPDATE instruction was not an IFILE or RIFILE name.

00060 A tab in a READ, WRITAB, READKS, or UPDATE instruction was greater than 249.

00061 A tab in a READ, WRITAB, READKS, or UPDATE instruction was zero. Note that if the value of an EQU'd tab is incorrectly specified the compiler generates a value of zero for the tab, and each use of that tab will generate this error.

00062 A character following an operand in the operand string of a READ, WRITE, WRITAB, READKS, or UPDATE instruction was not a space, comma, semicolon, or colon. If the instruction is a WRITAB or UPDATE instruction a semicolon is assumed.

00063 The character following the second operand of a WEOF instruction was not a space.

00064 The character following the second operand of a WRITAB instruction was not a semicolon.

00065 The first operand of an OPEN instruction was not a FILE, RFILE, IFILE or RIFILE name or the first operand of a PREPARE instruction was not a FILE or RFILE name.

00066 The first operand of a PREPARE instruction was an IFILE or RIFILE name.

There is no provision within the DATASHARE 3 INTERPRETER

for the creation of an indexed-sequential file. The file must first exist and be indexed by means of the INDEX program before the file may be opened by the OPEN instruction and accessed, increased, or decreased by means of the READ, WRITE, WRITAB, WEOF, READKS, UPDATE, and DELETE instructions.

00067 The character following the first operand of an OPEN or PREPARE instruction was not a comma.

00070 The character following the second operand of an OPEN or PREPARE instruction was not a space.

00071 The second operand of an OPEN or PREPARE instruction was not a string variable name or a literal.

00072 The end-of-line was encountered before a first operand was encountered in a CLOSE instruction.

00073 The first operand of a CLOSE instruction was not a FILE, RFILE, IFILE or RIFILE name.

00074 The character following the operand of a CLOSE instruction was not a space.

00075 A character following a list operand in a STORE, LOAD, or BRANCH instruction was not a comma, colon, or space.

00076 The first operand of a CLOCK instruction was not TIME, DAY, or YEAR.

00077 A comma or the preposition TO was not used between the first and second operands of the CLOCK instruction.

00100 The second operand of a CLOCK instruction was not a string variable.

00101 The character following the second operand of a CLOCK instruction was not a space.

00102 The first operand of an INSERT or DELETE instruction was not an IFILE or RIFILE name.

00103 The character following the first operand of an INSERT or DELETE instruction was not a comma.

00104 The second operand of an INSERT or DELETE instruction was not a string variable name.

00105 The character following the second operand of an INSERT or DELETE instruction was not a space.

00106 An alphabetic character string where a preposition should have been was not recognized as a preposition: BY, TO, OF, FROM, or INTO, or

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

00107 An EQUATE directive was given after an executable instruction was specified.

00110 An EQUATE directive was given but no label was specified.

00111 The first character of the operand of an EQUATE directive was not 1 thru 9. A first character of 0 implies an octal number which is not allowed in the EQUATE directive.

00113 The value specified for an EQUATE directive was not from 1 thru 249.

00114 The file specified in an INCLUDE directive was not found on disk.

00115 The character after the first operand of a DIM instruction was not a space.

00116 The operand value of a DIM instruction was greater than 127.

00117 For an INIT instruction or an instruction using a string literal:

No operand was found, or

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

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

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

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

A quoted string of greater than 127 characters was specified.

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

The character following the ending double-quote symbol of a quoted string was not a comma or a space.

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

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

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

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

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

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

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

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

The following applies if a quoted string was specified:

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

There were no digits specified, or

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

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

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

00127 An operand was not a quoted item, a number, or a label.

00130 The second character after the opening double-quote symbol in the operand of a CMOVE or CMATCH instruction was not a double-quote symbol. The forcing character does not apply in these two instructions because it is not necessary.

00131 For an instruction using a literal: the character after the ending double-quote symbol was not a space or comma.

00132 An octal number was specified but the number was not in the range 0 thru 037 inclusive.

00137 Internal compiler error.

00141 The operand of a PI instruction was not an unquoted numeric literal with a value of 1 through 20.

00143 Restricted error.

00144 Restricted error.

00145 Restricted error.

00146 CHECK10, CHECK11, REPLACE operand 1 not a svar.

00147 CK10, CK11, REP operand 2 not a svar or slit.

00150 SEARCH operand 1 not svar or nvar.

00151 SEARCH operand 2 not same type as operand 1.

00152 List length not nvar in SEARCH.

00153 Index not numeric in SEARCH.

00154 WRITE control was not *MP.

00155 WRITE control was not *ZF.

00156 Restricted error.

00157 COMLST in common area not allowed.

00160 Length specified in COMLST declaration was greater than 64 less than 1.

00161 The variable in COMCLR or COMTST instruction not followed by blank.

00162 No operands were specified in COMCLR, COMTST, SEND, or RECV or ACALL instruction.

00163 The first operand in COMCLR, COMTST, SEND, or RECV was not a COMLST variable.

00164 Missing ',' after COMLST variable or ';' after routing variable in SEND or RECV instruction.

00165 Missing routing variable in SEND or RECV instruction.

00166 Routing variable is not a string variable.

00167 Premature end of line.

00170 Invalid expression type in list used with SEND, RECV or ACALL.

00171 Unknown separator or character.

APPENDIX F. INDEX FILE SIZE COMPUTATION

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

For the following discussion the following definitions will be used:

NR = Number of logical records to be indexed.

KL = Key length (number of characters per key).

NS(i)= Number of disk sectors for the i th level of the tree.

NKSL = Number of keys per disk sector for the lowest level of the tree.

NKS = Number of keys per disk sector for other than the lowest level of the tree.

The number of sectors, NS(1) required for the lowest level of the tree is:

NKSL = 250/(KL+7) (discard remainder)

NS(1)= NR/NKSL (round up)

If NS(1)>1, then perform the following iterative calculation (i=2,3, etc), otherwise go to (2).

NKS = 250/(KL+3) (discard remainder)

(1) NS(i)= NS(i-1)/NKS (round up)

If NS(i)>1, then i=i+1 and go to (1) and repeat the process.

If NS(i)=1, then the iterative computation is complete and the total number of sectors (TNS) required for the complete index structure is:

(2) TNS = NS(1)+NS(2)+...+NS(i)

Note that this computation yields a maximum number of disk sectors required for the complete index structure and that the actual number used may be less.

Example:

NR = 10000 (10000 logical records to be indexed)

KL = 10 (key length is 10 characters)

Now the following computations are performed:

NKSL = 250/(KL+7) = 250/(10+7) = 14.71 = 14

NS(1)= NR/NKSL = 10000/14 = 714.29 = 715

The lowest level of the index tree requires 715 sectors. Since NS(1)>1, then i=i+1 = 2. Proceeding with the computation:

NKS = 250/(KL+3) = 250/(10+3) = 250/13 = 19.23 = 19

NS(2)= NS(i-1)/NKS = NS(1)/NKS = 715/19 = 37.63 = 38

The next highest level of the index tree requires 38 sectors. Since NS(2)>1, then i=i+1 = 3. Proceeding with the computation:

NS(3)= NS(i-1)/NKS = NS(2)/NKS = 38/19 = 2.00 = 2

The next highest level of the index tree requires 2 sectors. Since NS(3)>1, then i=i+1=4. Proceeding with the computation:

NS(4)= NS(i-1)/NKS = NS(3)/NKS = 2/19 = 0.11 = 1

The next highest level of the index tree requires 1 sector. Since NS(i)=1 has been reached, the computation is complete and we can now sum the total number of sectors (TNS) required.

TNS = NS(1)+NS(2)+NS(3)+NS(4)

TNS = 715+38+2+1 = 756

Therefore 756 sectors are required for the entire index tree.

APPENDIX G. SERIAL BELT PRINTER CONSIDERATIONS

Since the serial belt printer is connected to a 3600 terminal, there is no way that printer status information can be returned to the interpreter. This means that all timing considerations required by the printer must be handled by sending enough "pad" characters to satisfy the worst case print time. (A pad character is any character that will not be printed by the printer. For example, an octal 032 will work quite well as a pad character.)

Calculating the number of pad characters can sometimes be confusing. The following discussion will hopefully eliminate some of the confusion.

SIMPLE BUT SLOWER SOLUTION

The simplest way to handle the timing considerations is to use a *W list control in every DISPLAY statement that will cause printing on the belt printer. The one second pause will provide more than enough time for the printer to print a line.

MORE DIFICULT SOLUTION

The belt printer requires that a certain minimum of characters be sent per line. If less than this minimum is sent, the printer can become very confused and erratic. This minimum number of characters that must be sent is dependent on both the baud rate of the 3600 to which it is attached and also, the length of the line being sent.

The following table shows the smallest line that can be sent to the printer.

Baud Rate	Line Length: less than 40	Line Length: greater than or equal to 40
110 (11 bits/char)	3	N/A
110 (10 bits/char)	3	N/A
150	4	N/A
220 (11 bits/char)	5	N/A
220 (10 bits/char)	6	N/A
300	7	N/A
600	14	N/A
1200	28	56
2400	56	111
4800	111	221
9600	221	442

N/A indicates that timing does not need to be considered when using the indicated baud rate and line length.

Example: Let n represent the number of characters in the line to be printed. If the terminal to which the printer is connected is set to 1200 baud, then:

a) If n < 28, enough pads must be added to make n = 28.

b) If 28 < n < 40, no pads need to be added. The line may be printed "as is".

c) If 40 < n < 56, enough pads must be added to make n = 56.

d) If 56 < n, no pads need to be added.

To turn the printer on, so that anything displayed at the terminal will get printed, the 032 control character should be displayed. To turn the printer off, so that the terminal can be used without the printer; the 024 control character should be displayed.

TURNING THE PRINTER OFF

Lines are not printed by the serial belt printer until an 012

or 015 control is received by the printer. If the printer were never to receive an 012 or 015, no lines would get printed. The Databus DISPLAY statement normally furnishes these controls at the end of the line.

Consider the following DISPLAY statement:

```
DISPLAY   032,*W,"LINE TO BE PRINTED",024
```

This line will <u>not</u> be printed. The following sequence is sent to the terminal by this display statement. First, the printer is turned on. Second, the wait control is used to handle the timing considerations. Third, the line is displayed on the terminal and sent to the printer. Fourth, the printer is disconnected from the terminal. Fifth, a carriage return (015) and line feed (012) character are sent to the terminal. Note that neither the 015 nor the 012 got sent to the printer because it was turned off before the controls were sent.

The simplest way to solve this problem is to turn the printer on and off in different DISPLAY statements from the one used to display data at the terminal. Each DISPLAY statement to be sent to the printer does <u>not</u> need to turn the printer on and then turn it off.

Example:

```
FILE      FILE
SEQ       FORM      "-1"
LINE      DIM       80
.
          OPEN      FILE,"DATA"
          DISPLAY   032
          GOTO      BEGIN
.
 LOOP      DISPLAY   *W,*R,*P1:12,*+,LINE
BEGIN     READ      FILE,SEQ;LINE
          GOTO      LOOP IF NOT OVER
.
          DISPLAY   024
          STOP
```