DSM-11
Language Reference Manual

Order Number AA-H797B-TC

March, 1984

This document describes the syntax and elements of DSM-11 language.

This is a revised manual.

Operating System: DSM-11 Version 3

Software: DSM-11 Version 3

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid-Atlantic Region

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884–6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640–5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734–4915

digital equipment corporation • marlboro, massachusetts

First Printing, October, 1980
Revised, March, 1984

CONTENTS

Part 1: Language Syntax

CHAPTER 1 DSM-11

1.1 INTRODUCTION TO DSM-11 1-1
1.2 THE DSM-11 CHARACTER SET 1-1
1.2.1 Uppercase And Lowercase Characters 1-2
1.2.2 Nonprinting Control Characters 1-3
1.3 STATEMENT STRUCTURE 1-4
1.3.1 Commands 1-4
1.3.2 Command Abbreviations 1-4
1.3.3 Arguments 1-5
1.3.4 Argument Spacing 1-5
1.3.5 Argument Lists 1-6
1.3.6 Statement Spacing 1-6
1.4 LINE STRUCTURE 1-7
1.4.1 Command Lines 1-7
1.4.2 Routine Lines 1-8
1.4.2.1 Line Labels 1-8
1.4.2.2 The TAB Character 1-9
1.4.2.3 Comments 1-9
1.5 ROUTINE STRUCTURE 1-10
1.6 REFERENCES 1-11
1.6.1 Line References 1-11
1.6.2 Entry References 1-12
1.6.3 Line Specifications 1-12
1.7 EXTENSIONS TO ROUTINE STRUCTURE 1-13

1.7.1 Overview 1-13
1.7.2 Execution Levels 1-14
1.7.3 Line Structure For Block Structuring 1-15
1.7.4 Routine Structure And Execution 1-15

CHAPTER 2 DSM-11 EXPRESSION ELEMENTS

2.1 OVERVIEW OF DSM-11 EXPRESSION ELEMENTS 2-1
2.2 FUNCTIONS 2-2
2.3 LITERALS 2-3
2.3.1 Numeric Literals 2-3
2.3.1.1 Integer Literals 2-4
2.3.1.2 Decimal Numeric Literals 2-4
2.3.1.3 Exponential Notation 2-6
2.3.2 String Literals 2-7
2.4 VARIABLES 2-8
2.4.1 Local Variables 2-9
2.4.2 Global Variables 2-11
2.4.3 Extended Global References 2-12
2.4.4 Array Structure 2-13
2.4.5 Naked References 2-17
2.4.6 Special Variables 2-19
2.5 UNARY OPERATORS 2-20
2.6 BINARY OPERATORS 2-20
2.6.1 Arithmetic Operators 2-21
2.6.2 Numeric Relational Operators 2-22
2.6.3 String Relational Operators 2-23
2.6.4 The String Operator 2-24
2.6.5 Logical Operators 2-24
2.6.6 Indirection Operator 2-25
2.6.6.1 Argument Indirection 2-26
2.6.6.2 Name Indirection 2-26
2.6.6.3 Pattern Indirection 2-27
2.6.6.4 Partial Indirection 2-28
2.6.7 Formatting Characters 2-29
2.6.7.1 The Form-Feed Character 2-30
2.6.7.2 The Carriage-Return/Line-Feed Character 2-30
2.6.7.3 The Horizontal-Tabulation Character 2-30
2.7 EXPRESSION EVALUATION 2-31
2.7.1 String Expressions 2-33
2.7.2 Numeric Expressions 2-34
2.7.3 Truth-Valued Expressions 2-34

2.7.4 Postconditional Expressions 2-34
2.7.4.1 The Command Postconditional Expression 2-34
2.7.4.2 The Argument Postconditional Expression 2-35
2.7.4.3 Naked References And Postconditional Expressions 2-36
2.7.5 Timeout Expressions 2-36

Part 2: Language Reference

CHAPTER 3 DSM-11 OPERATORS

3.1 INTRODUCTION TO DSM-11 OPERATORS 3-1
3.2 OPERATOR DESCRIPTIONS 3-1
Binary ADD (+) 3-2
Binary AND (&) 3-4
Binary CONCATENATE (_) 3-6
Binary CONTAINS ([) 3-7
Binary DIVIDE (/) 3-9
Binary EQUALS (=) 3-11
Binary FOLLOWS (]) 3-14
Binary GREATER THAN (>) 3-17
Binary INCLUSIVE OR (!) 3-19
Binary INTEGER DIVIDE (\) 3-21
Binary LESS THAN (<) 3-23
Binary MODULO (#) 3-25
Binary MULTIPLY (*) 3-28
Binary PATTERN MATCH (?) 3-30
Binary SUBTRACT (-) 3-34
INDIRECTION (@) 3-36
Unary MINUS (-) 3-40
Unary NOT(') 3-42
Unary PLUS (+) 3-45

CHAPTER 4 DSM-11 ANSI STANDARD COMMANDS

4.1 INTRODUCTION TO DSM-11 ANSI STANDARD COMMANDS 4-1
4.2 COMMAND DESCRIPTION 4-2
BREAK 4-3
CLOSE 4-6
DO 4-8
ELSE 4-12
FOR 4-14
GOTO 4-19
HALT 4-22

HANG 4-23
IF 4-25
JOB 4-28
KILL 4-31
LOCK 4-34
NEW 4-39
OPEN 4-43
QUIT 4-46
READ 4-48
SET 4-54
USE 4-61
VIEW 4-64
WRITE 4-71
XECUTE 4-74

CHAPTER 5 DSM-11 EXTENDED COMMANDS

5.1 INTRODUCTION TO DSM-11 EXTENDED COMMANDS 5-1
5.2 EXTENDED COMMAND DESCRIPTION 5-2
ZALLOCATE 5-3
ZBREAK 5-7
ZDEALLOCATE 5-9
ZGO 5-11
ZINSERT 5-13
ZLOAD 5-18
ZPRINT 5-20
ZQUIT 5-23
ZREMOVE 5-26
ZSAVE 5-29
ZTRAP 5-31
ZUSE 5-33
ZWRITE 5-35

CHAPTER 6 DSM-11 FUNCTIONS

6.1 INTRODUCTION TO DSM-11 FUNCTIONS 6-1
6.2 FUNCTION DESCRIPTIONS 6-2
$ASCII 6-3
$CHAR 6-5
$DATA 6-8
$EXTRACT 6-10
$FIND 6-13
$JUSTIFY 6-16
$LENGTH 6-19
$NEXT 6-21

$ORDER....6-23
$PIECE....6-26
$RANDOM....6-30
$SELECT....6-31
$TEXT....6-33
$VIEW....6-35
$ZCALL....6-38
$ZNEXT....6-39
$ZORDER....6-41
$ZSORT....6-45
$ZUCI....6-47

CHAPTER 7 DSM-11 SPECIAL VARIABLES

7.1 INTRODUCTION TO DSM-11 SPECIAL VARIABLES....7-1
7.2 SPECIAL VARIABLE DESCRIPTIONS....7-2
$HOROLOG....7-3
$IO....7-4
$JOB....7-5
$STORAGE....7-6
$TEST....7-7
$X....7-9
$Y....7-10
$ZA....7-11
$ZB....7-12
$ZBREAK....7-13
$ZERROR....7-17
$ZORDER....7-18
$ZREFERENCE....7-20
$ZTRAP....7-21
$ZVERSION....7-23

APPENDIX A ASCII CHARACTER LANGUAGE SET

APPENDIX B DSM-11 LANGUAGE SUMMARY

B.1 DSM-11 SYNTAX SUMMARY....B-1
B.2 DSM-11 ANSI STANDARD COMMANDS....B-3
B.3 EXTENDED DSM-11 COMMANDS....B-7
B.4 DSM-11 FUNCTIONS....B-10
B.5 DSM-11 SPECIAL VARIABLES....B-12
B.6 DSM-11 OPERATORS....B-13
B.7 DSM-11 SPECIAL SYMBOLS....B-14

FIGURES

2-1 Global Array Structure....2-13
2-2 Global Array (Naked References)....2-17
4-1 Locked Global....4-36
5-1 Flow of Control with ZQUIT....5-24
6-1 A $ORDER, $ZSORT, or $NEXT Array Scan....6-42
6-2 A $ZORDER or $ZNEXT Array Scan....6-43

TABLES

2-1 DSM-11 Unary Operators....2-20
2-2 DSM-11 Arithmetic Operators....2-22
2-3 Numeric Relational Operators....2-22
2-4 String Relational Operators....2-23
2-5 Logical Operators....2-24
2-6 Truth Table....2-25
2-7 Formatting Characters....2-29
4-1 VIEW Argument Values....4-68
6-1 $VIEW Argument Values....6-36

Acknowledgement

DIGITAL Standard MUMPS is an extension of the ANSI Standard Specification (X11.1-proposed for final approval in 1984) for the Massachusetts General Hospital Utility Multi-Programming System (MUMPS). MUMPS was originally developed at the Laboratory of Computer Science at Massachusetts General Hospital and was supported by grant HS00240 from the National Center for Health Services Research and Development.

Preface

MANUAL OBJECTIVES

This manual describes the language elements of Digital Standard MUMPS for the PDP-11 (DSM-11). It does not discuss terminal usage or provide information related to the DSM-11 operating system. You can find such information in the *DSM-11 User's Guide*.

INTENDED AUDIENCE

This manual is intended as a reference for DSM-11 users who are familiar with higher level languages and programming techniques. Because this manual is designed for experienced programmers, it does not present information tutorially.

MANUAL STRUCTURE

The *DSM-11 Language Reference Manual* is divided into two parts:

1. Part 1: Language Syntax

   Part One consists of Chapters 1 and 2. It describes DSM-11 language elements and syntax.

2. Part 2: Language Reference

   Part Two consists of Chapters 3 through 7. It contains descriptions of the DSM-11 operators, commands, functions, and special variables.

   The material in this part is arranged in alphabetical order by element type. All descriptions use the following organizational structure:

PURPOSE:	This section explains briefly what the language element does.
FORM:	This section shows the form of the language element.
EXPLANATION:	This section explains the forms of the language element in more detail.
COMMENTS:	This section describes any special considerations you should keep in mind while using the language element. (Not all language-element descriptions have a comments section.)
RELATED:	This section lists all language elements or concepts related to the language element being described. (Not all language-element descriptions have a related section.)
EXAMPLES:	This section presents examples of the language element. The examples range from single lines to short routine fragments.

The manual also contains two appendixes and a glossary. Appendix A lists the ASCII character set; Appendix B presents a summary of DSM-11 language elements. The glossary defines terms used in the manual.

RELATED DOCUMENTS

This manual is part of the DSM-11 documentation set. This set also includes:

The *DSM-11 BISYNC Programmer's Guide* (AA-V602A-TC)

The *DSM-11 Summary* (AV-H798B-TC)

The *DSM-11 User's Guide* (AA-H799B-TC)

The *DSM-11 XDT Reference Manual* (AA-J701A-TC)

The *Introduction to DSM* (AA-K676A-TK)

DOCUMENTATION CONVENTIONS

This manual uses the following documentation conventions and symbols:

Convention	Meaning
(CTRL/X)	A key (represented here by x) typed while the CTRL key is pressed.
(ESC)	The escape key.
(PERIOD)	The period key.
(RET)	The carriage return key.
(SP)	The space bar.
(TAB)	The TAB key ((CTRL/I) on some terminals).
lowercase	A language element or portion of a language element in example syntax.
{ }	The enclosed element is optional.
	A break in a series or elements in a series not shown in the manual. For example, this symbol is used to show how to structure multiple command arguments.
	A break between two illustrated lines of code in a routine example.
Contrasting Colors in Examples:	
	Black - indicates a noninteractive example, or system output in an interactive example
	Red - indicates user input in an interactive example

Part 1: Language Syntax

Chapter 1
DSM-11 Syntax

This chapter introduces the DSM-11 language and discusses its syntax and structure. It discusses the DSM-11 character set and details the structure of DSM-11 statements, lines, and routines.

1.1 Introduction to DSM-11

DSM stands for Digital Standard MUMPS. DSM-11 is Digital Equipment Corporation's implementation of Standard MUMPS (Massachusetts General Hospital Utility Multi-Programming System) for the PDP-11.

DSM-11 encompasses the American National Standards Institute's (ANSI) *Standard Specification for MUMPS* X11.1-1977, but DSM-11 also provides many extensions to that standard. Version 3 of DSM-11 also encompasses the most recent additions and changes to the language reflected in the 1984 version of the ANSI MUMPS standard.

1.2 The DSM-11 Character Set

DSM-11 stores data as strings of ASCII (American Standard Code for Information Interchange) characters as defined in ANSI Standard X3.4-1968. For command and control purposes, DSM-11 uses a 64-character subset of the 128-character ASCII set.

This subset consists of ASCII decimal values 32 through 95. These values include:

- The uppercase letters A through Z
- The numbers 0 through 9
- Symbolic characters

This subset and the lowercase alphabetic characters (ASCII values 32 through 126) are called the *ASCII graphic character set*. Graphic characters are those that the system can reproduce on your terminal or line printer as the characters they represent.

(See Appendix A for a list of the ASCII character set.)

1.2.1 Uppercase And Lowercase Characters

The ASCII characters with values of 97 through 122 are lowercase alphabetic characters. You can use uppercase or lowercase characters in DSM-11 language elements, variable names, routine names, and line labels. However, DSM-11 does not always recognize lowercase letters as the equivalent of uppercase letters. The following rules govern the interpretation of lowercase and uppercase letters in DSM-11.

1. Language Elements

   DSM-11 treats lowercase, uppercase, and mixed case commands, functions, special variables, and binary PATTERN MATCH characters as equivalents. For example, DSM-11 considers the following three commands identical:

   ```
   WRITE HELLO

   write HELLO

   WrITe HELLO
   ```

2. Local Variables

   DSM-11 considers lowercase, uppercase, and mixed case local variable names unique. DSM-11 considers the local variables used in the following examples as three separate variables:

   ```
   SET XX="HELLO"

   SET Xx="HELLO"

   SET xx="HELLO"
   ```

3. Global Variables

   DSM-11 treats lowercase, uppercase, and mixed case global variable names as unique. (Global variables are variables whose names are preceded by the circumflex character.) For example, DSM-11 treats the global variables in the following as three different variables:

```
SET ^XX="HELLO"

SET ^Xx="HELLO"

SET ^xx="HELLO"
```

4. Routine Names

   DSM-11 considers lowercase, uppercase, and mixed case routine names unique. Thus, the following three commands store three separate routines on disk:

```
ZSAVE TEST

ZSAVE test

ZSAVE Test
```

5. Line Labels

   DSM-11 considers lowercase, uppercase, and mixed case line labels unique. Thus, the following three commands direct control to three separate routine lines:

```
GOTO LABEL1

GOTO label1

GOTO Label1
```

In general, mixed uppercase and lowercase letters can be used in string literals, input data, and comments.

1.2.2 Nonprinting Control Characters

The first 32 characters of the ASCII set (decimal values 0 through 31) and the last character of the ASCII set (decimal value 127) are nonprinting characters. They have system control and editing functions.

1.3 Statement Structure

The basic element in DSM-11 is the *statement* . A statement specifies an operation for DSM-11 to perform. Each statement is made up of a command and, optionally, one or more arguments.

The general format for a statement is:

command (SP)*argument(s)*

The following sections describe commands and arguments.

1.3.1 Commands

Commands are names composed of alphabetic characters. Each command name is a mnemonic for the action the command performs.

DSM-11 has two types of commands:

- ANSI Standard commands
- Extended commands

ANSI Standard commands are specified in the *ANSI MUMPS Language Standard* and follow Standard usage. The Standard reserves the letters A through Y as initial letters in Standard command names.

As specified in the *ANSI MUMPS Language Standard*, extended commands are implementation-specific commands. The *ANSI MUMPS Language Standard* reserves the letter Z as the initial letter in extended-command names.

1.3.2 Command Abbreviations

When you use a command, you can enter either the full spelling or an abbreviation of the command name. For ANSI Standard commands, the abbreviation consists of the first letter of the command name.

For example, you can write the ANSI Standard command GOTO in either of the following ways:

GOTO(SP) *argument list*

G(SP) *argument list*

For extended commands, the abbreviation consists of the first two letters of the command name (the Z and the next sequential letter).

For example, you can write the extended command ZLOAD in either of the following ways:

ZLOAD[SP] *argument*

ZL[SP] *argument*

If two commands can be abbreviated to the same letter (HALT and HANG), DSM-11 distinguishes which command you mean by the presence or absence of an argument.

1.3.3 Arguments

Arguments define and control the action of the command to which you append them. The nature of an argument depends on the command with which you use it. Consider the following statement:

```
GOTO A1
```

In this statement, the argument A1 is a *line label*. (See Section 1.4.2.1 for more information on line labels.)

Consider also the following statement:

```
WRITE A+4
```

In this statement, the argument A+4 is an *expression* composed of a *local variable name* (A), an *operator* (+), and a *numeric literal* (4). (See Chapter 2 for more information on expressions and expression elements.)

Some commands always take arguments. Others never take arguments. Some commands take arguments under certain circumstances. Commands -- such as KILL -- that take arguments only under certain circumstances have a different meaning depending on whether they do or do not have an argument.

1.3.4 Argument Spacing

Spacing is significant in DSM-11. If you are entering a command that takes an argument, you must separate that command from its argument with one space. If you enter any other character in place of the space or enter more than one space, DSM-11 reports an error.

For example, the following statement is valid:

```
DO SP B3
```

But the following statements are not:

```
DO SP SP )B3
```

```
DOB3
```

1.3.5 Argument Lists

Argument lists consist of more than one argument appended to a single command. They are a shorthand method of performing the same action on a number of elements.

For example, the following statements are equivalent:

```
SET I=6,B=10,C=11
```

```
SET I=6 SET B=10 SET C=11
```

You must separate multiple arguments for one command with commas and no intervening spaces. For example, the following statement is valid:

```
WRITE A(I),B(I),C(I)
```

However, the following statement is not:

```
WRITE SP A(I), SP B(I), SP C(I)
```

1.3.6 Statement Spacing

You can enter more than one statement on a line. But to do so, you must use spaces to separate the statements. If you enter a command that takes arguments, you must separate its last argument from the command that follows with one or more spaces, for example:

```
SET SP A=0 SP GOTO SP B1
```

If you enter a command that does not take arguments, you must separate it from the command that follows it with two or more spaces. (DSM-11 considers the first space as part of the command and the following spaces as the delimiter between the command and the next command.) For example:

```
ELSE SP SP GOTO SP B1
```

If a command with arguments comes at the end of the line, do not enter any spaces between the last argument and the carriage return, for example:

SET(SP)J=0 (RET)

If a command without an argument immediately precedes the end of the line, do not enter any spaces between the command and the carriage return, for example:

IF ANS=(SP)QUIT(RET)

1.4 Line Structure

All DSM-11 code is organized in *lines*. Each line contains one or more statements and ends with a carriage return.

The DSM-11 line is not limited to one physical terminal line. A DSM-11 line can contain as many as 255 characters, including the line label but excluding the terminating carriage return.

DSM-11 recognizes two types of lines:

1. Command lines
2. Routine lines

1.4.1 Command Lines

A *command line* consists of one or more statements you enter for immediate execution. After you type a carriage return, DSM-11 interprets the statements and acts upon them.

DSM-11 recognizes that a line is a command line by its format. The general format of a command line with a comment is:

{*command*}(SP){*arg list*}{...}{(SP)...(SP)};*comment* (RET)

If a command line has no comment at the end of it, there should be no spaces at the end of the line:

{*command*} (SP){*arg list*}{...}(RET)

where:

command	is any valid command
arg list	is an argument list
comment	is a comment (See Section 1.4.2.3 for more information on comments.)

You must enter the first command immediately after the DSM-11 pr Do not enter a space character (or other character) between the prompt and the first letter of the command name.

1.4.2 Routine Lines

A *routine line* consists of one or more statements entered for later execution. After you type a carriage return at the end of the line, DSM-11 stores the line and displays its prompt for more input.

DSM-11 recognizes a line as a routine line by its format. A routine line must begin with a TAB character or a line label and a TAB character. The TAB must be followed by a command or by one or more spaces followed by a command. The general format for a routine line is:

{*label*}TAB{SP...SP}{*command*}SP{*arg list*}{...}{SP...SP};*comment*

where:

label	is a line label
command	is any valid command
arg list	is an argument list
comment	is a comment

If there is no comment at the end of the line, there should be trailing spaces at the end of the line.

1.4.2.1 Line Labels—The *line label* is an optional name that identifies the line to which it is prefixed. Each line label should contain no more than eight characters. The characters can be alphabetic characters, the digits 0 through 9, or the percent (%) character.

The format for a line label can be one of the following:

alpha alpha/digit ...

%alpha/digit ...

digit digit ...

where:

alpha	is any alphabetic character
digit	is any digit character
alpha/digit	alpha is any alphabetic character or any digit character

If the first character in the line label is an alphabetic character a percent character, the other characters can be any combination of alphabetic characters or digit characters. If the first character in the line label is a digit, the other characters must be digits.

DSM-11 considers leading zeros significant in digit line labels. That is, the following labels are different:

01

1

A particular line label should appear only once in a routine. Although the identical line label may appear more than once in a routine, DO, GOTO, and $TEXT always find the first appearance of that line label.

1.4.2.2 The TAB Character—The TAB character (ASCII decimal value 9) is one of the nonprinting control characters. It causes a skip to the next TAB stop. (CTRL/I also performs the same function.)

In DSM-11, TAB (or CTRL/I) indicates that a line is a routine line. If you include a line label on the routine line, you must use the TAB character as the first character following the line label.

If you do not include a line label on the routine line, you must use the TAB character as the first character on the line.

1.4.2.3 Comments—The last element in a routine line is an optional *comment* (identified by a preceding semicolon). The comment is usually a brief explanation of the purpose of a routine or of a routine line.

You can use any ASCII graphic character in a comment. You cannot use any of the nonprinting ASCII control characters, the escape character, the form-feed character, the carriage return, or the CTRL character in combination with any other character.

The format for a commented routine line is:

{*label*}TAB{...}{SP... SP}; *comment*RET

You can use an entire DSM-11 line for a comment. In this case, enter the identifying semicolon and the comment directly after the TAB character. For example:

```
BILL3 ;SMITH:11-JAN-80:BILLING ACCOUNT INQUIRY
```

You can also append comments to routine lines in any of the following ways:

- Following directly after the last argument or command
- Separated from the argument list of the last command by one or more spaces
- Separated from an argumentless command by two or more spaces

In the following example routine fragment, each addition of a comment to a line is valid:

```
SET A=0,I=0,CNT=1; INITIALIZE COUNTERS
SET NAME=""; INITIALIZE NAME VARIABLE


.
END WRITE !,"FINISHED" QUIT  ;EXIT ROUTINE
```

1.5 Routine Structure

Each *routine* is a sequence of lines that you save, load, and execute as a unit. Each routine must have a name of from one to eight characters. DSM-11 uses only the first eight characters assigned as the name of the routine.

The format for a routine name can be one of the following:

alpha alpha/digit ...

% alpha/digit ...

where:

alpha	is any alphabetic character
alpha/digit	is any alphabetic character or character
%	is the percent character

The first character in a routine name must be an alphabetic char or the percent character. (By convention, only library routines use the percent character in their names.) The remaining characters in the name can be either alphabetic or digit characters.

1.6 References

The term *reference* describes several methods of using line labels and routine names to control the flow of processing or to manipulate routines. DSM-11 recognizes three types of references:

1. Line references
2. Entry references
3. Line specifications

1.6.1 Line References

The *line reference* is a means of specifying a line within the routine currently in memory. You can use line references as arguments with commands such as ZINSERT, ZPRINT, and ZREMOVE to edit or write the current routine.

The simplest form of line reference is a line label, for example:

```
GOTO PART2
```

You can also reference any line that does not have a label by using an *offset* from a prior line that has a label. The format for such a line reference is:

label + *offset*

where:

label	is the label of a prior line
offset	is an integer value that specifies how many lines there are from the labeled line to the line being referenced

For example, the following line reference refers to the fifth after the line labeled A.

A + 5

Offsets can be zero. Such references, however, are always equal to the labeled line. For example the following line reference refers to the line labeled A.

A + 0

Offset references are never negative. You cannot reference an unlabeled line that precedes the labeled line.

When you use a line reference as an argument with certain editing commands, DSM-11 uses the line reference to position an implicit, internal line pointer into the routine currently in memory. DSM-11 sets the line pointer at the point following the end of the referenced line and before the beginning of the next line.

1.6.2 Entry References

Entry references are a type of reference used with the control commands DO and GOTO. They are a means of directing control to a line in the routine in memory or to any other routine.

The format for such an entry reference is:

{*label*{+*offset*}}{^*routine name*}

If the entry reference includes only a routine name, (preceded by a circumflex), DSM-11 loads the specified routine and begins execution at the first executable line. If the entry reference also includes a line label or line label and offset, DSM-11 loads the specified routine and begins execution at the specified line.

If no line in the routine has a label that matches the spelling of the label in the entry reference or if the integer offset is greater than the number of lines that follow the labeled line in the routine, DSM-11 reports an error.

1.6.3 Line Specifications

With the commands ZINSERT, ZPRINT, and ZREMOVE and the function $TEXT, you can also reference a line with a *line specification*. A line specification is an integer value preceded by a plus sign that denotes the sequential position of a line within the routine currently in memory.

For example, you can reference the first line in a routine n lines long with:

+1

You can reference the third line with:

+3

You can reference the last line with:

+n

When you use a line specification as an argument, DSM-11 uses the line specification to position an implicit, internal line pointer within the current routine. DSM-11 sets the line pointer at the point following the end of the referenced line and before the beginning of the next line in sequence.

1.7 Extensions to Routine Structure

This section describes a new feature of DSM-11 for Version 3: block structuring. Block structuring is an extension of the language and represents a feature not yet accepted in the ANSI MUMPS Standard.

1.7.1 Overview

Block structuring uses an argumentless DO command to transfer control to an indented block of routine lines which are treated as a subroutine. The indented block of lines must immediately follow the line with the argumentless DO in it. After these lines are executed, control returns to the previous level of indentation. The indented block of lines is referred to as an execution block. Periods are used to indicate the level of indentation.

For example, the following is a simple block-structured routine:

```
TEST  W !,"LEVEL A" F I=1:1:2 D
      .W !,"LEVEL B" F J=1:1:2 D
      ..W !,"LEVEL C"
      W !,"LEVEL A AGAIN"
```

This routine gives the result:

```
LEVEL A
LEVEL B
LEVEL C
LEVEL C
LEVEL B
LEVEL C
LEVEL C
LEVEL A AGAIN
```

The reason for using block structuring is to extend the scope of FOR, IF, and ELSE statements without creating subroutines that are called only once. This results in a large number of labels and a confusing visual flow. These subroutines are often needed because of the single-line scope of the FOR, IF, and ELSE commands. Using block structuring results in a more logical and sequential flow to the program.

The following is an example routine for student records:

```
TEST    READ !,"Enter name: ",NAM
        FOR I=1:1 DO QUIT:SUB=""
        .READ !,"Enter subject: ",SUB QUIT:SUB=""
        .SET STR= ""
        .FOR MON=1:1:9 DO ..WRITE !,"Enter grade for month:",MON,":" READ G
        ..WRITE !,"Enter days absent for month: ",J,": READ A
        ..SET STR=STR_"G_":"_A_";"
        .SET ^DATA(NAM,SUB)=STR
```

When this routine is set up without structuring, it looks like this:

```
TEST    READ !,"Enter name: ",NAM
        FOR I=1:1 DO LABEL1 QUIT:SUB=""
        QUIT ;End of routine
LABEL1  READ !,"Enter subject: ",SUB QUIT:SUB=""
        SET STR=""
        FOR MON=1:1:9 DO LABEL2
        SET ^DATA(NAM,SUB)=STR
        QUIT
LABEL2  WRITE !,"Enter grade for month: ",MON,":" READ G
        WRITE !,"Enter days absent for month: ",J,":" READ A
        SET STR=STR_G_":"_A_";"
        QUIT
```

1.7.2 Execution Levels

The first line of the routine is the first, or lowest, level of execution. More deeply indented blocks represent higher levels of execution. Thus, the following line in the first example is the third level of execution and is a higher execution level than the other levels.

```
..W !,LEVEL C
```

Note that this line is indented with two periods after the tab stop. The level of the execution block is defined as the number of periods after the tab and before the command plus one. Thus, a tab followed by two periods indicates a third-level execution block.

1.7.3 Line Structure For Block Structuring

Execution blocks are initiated by an argumentless DO in the preceding (lower level) line. The structure of this initiating line is as follows:

{*label*}[TAB]{*command*}[SP]{*arg list*}{...}DO[SP][SP]
{*command*}[SP]{*arg list*}{...}

where *label* , *command* , *arg list*, and *comment* are defined in Section 1.4. Note that this line represents one continuous line even though it must be broken into two lines in order to fit on this page.

A line in an indented block (subroutine) has the following structure:

[TAB][PERIOD]{[PERIOD]...[PERIOD]}{*command*}[SP]{*arg list*}{...}DO[SP]
[SP]{*command*}[SP]{*arg list*}{...}

This line may itself contain an argumentless DO that initiates yet another higher level execution block.

1.7.4 Routine Structure And Execution

All indented lines that are intended to be in the same execution block must be indented with the same or greater number of periods. They cannot be interrupted by lines of a lower level, for example:

```
Label1  Line 1
        .Line 2
        .Line 3
        ..Line 4
        ..Line 5
        .Line 6
        ..Line 7
        ..Line 8
        Line 9
```

This routine does not perform as expected if lines 4,5,7, and 8 are intended to be part of the same execution block. DSM-11 regards lines 4 and 5 as a separate execution block initiated by an argumentless DO command in line 3; and lines 7 and 8, as an execution block initiated by a argumentless DO in line 6.

Execution within a block proceeds sequentially from line to line within the same level. Execution of the block terminates when a QUIT command is reached or when a line of a lower level is reached. Control passes to the next command in the initiating line following the argumentless DO command. If

the DO command is at the end of the initiating line, control passes to the next sequential line at the same or a lower level.

If lines of a higher level occur in a routine (but are not called by an argumentless DO command), then they are skipped over.

Blocks can be no more than one level higher than the preceding level. A line introduced by two additional periods (two higher levels) is ignored, even if the argumentless DO command is used correctly in the initiating line, for example:

```
Label1  Line 1
        Line 2
        ..Line 3
        ..Line 4
        Line 5
```

In this routine, lines 3 and 4 are ignored.

The GOTO command cannot be used to transfer between different level execution blocks within a routine. The GOTO command can be used to transfer control within the same execution block.

Chapter 2
DSM-11 Expression Elements

This chapter describes DSM-11 expression elements and expression evaluation.

2.1 Overview of DSM-11 Expression Elements

The basic unit in all DSM-11 arguments is the expression. Expressions are character strings that, when executed, create a value.

All expressions must contain at least one element called an *expression atom*. An expression atom can be:

- A function
- A literal
- A variable
- Another expression atom preceded by a unary operator

An expression can be composed of a single expression atom. For example, consider the following statements:

```
WRITE $L(A)
WRITE "1800 HOURS"
WRITE X
WRITE "+1800 HOURS"
```

In these statements, the arguments $L(A) (a function), "1800 HOURS" (a literal), X (a variable), and +"1800 HOURS" (another expression atom preceded by a unary operator) are expressions consisting of a single expression atom.

An expression can also be composed of a series of expression atoms separated by *binary operators*. Binary operators are special characters that indicate operations to be performed between two expression atoms.

In producing a result, DSM-11 does not change the value(s) of the original expression(s). It simply creates a new value based on the value(s) of the expression(s) and the operation indicated by the binary operator.

The following sections discuss the four basic elements in expression atoms (functions, literals, variables, and unary operators), and binary operators in more detail.

2.2 Functions

A *function* is an operation that returns a single value. Each function consists of a function name (an alphabetic word preceded by a dollar sign) followed by one or more arguments enclosed in parentheses.

The syntax of the function and its argument is as follows:

$*function* (*arg list*)

where:

function	is a DSM-11 function name
arg list	is the accompanying argument or argument list
( )	are the parentheses in which you must enclose the argument or argument list

The function name is a mnemonic for the operation the fun performs. DSM-11 has two types of functions:

- ANSI Standard functions
- Extended functions

ANSI Standard functions are specified by the *ANSI MUMPS Language Standard* and follow Standard usage. The Standard reserves the characters $A through $Y as the initial characters in ANSI Standard function names.

[illegible] specified in the *ANSI MUMPS Language Standard*, extended functions are implementation-specific additions to the language. Extended function names start with the characters: $Z

When you use an ANSI Standard function, you can enter the full spelling of the function name or an abbreviation consisting of the first two characters (the $ and the first letter of the function name). For example, you can write the ANSI Standard function $DATA as either of the following:

$DATA(*argument*)

$D(*argument*)

When you use an extended function, you can enter the full spelling of the function name or an abbreviation consisting of the first three characters (the $, the Z, and the first letter of the function name). For example, you can write the extended function $ZORDER in either of the following ways:

$ZORDER(*argument*)

$ZO(*argument*)

Function arguments are the values that DSM-11 uses to derive the value it creates by the operation. You must enclose the function argument or argument list in parentheses. You must not separate the function from the open parenthesis by any spaces. If you use multiple arguments, separate adjacent arguments by a comma and no spaces.

For example, you must write the function $JUSTIFY with the arguments X, 7, and 4 in either of two ways:

```
$JUSTIFY(X,7,4)

$J(X,7,4)
```

2.3 LITERALS

Literals are series of characters placed in a routine or command line that never change value. DSM-11 recognizes two types of literals:

- Numeric literals
- String literals

2.3.1 Numeric Literals

Numeric literals (also called *constants*) are strings that DSM-11 evaluates as numbers. DSM-11 treats as a number any string that contains:

- The digits zero through nine

- The Unary MINUS operator
- The Unary PLUS operator
- The period or decimal point character
- The letter E

DSM-11 recognizes both integer and decimal numeric literals.

2.3.1.1 Integer Literals—An integer literal is a series of one or more digits optionally preceded by a Unary MINUS or Unary PLUS operator, for example:

-18

235001

12367444

Each positive integer literal you use should contain only digits. It should not contain a decimal point.

Positive integer literals do not need a preceding Unary PLUS operator or leading zeros. DSM-11 discards Unary PLUS operators and leading zeros in arithmetic operations, for example:

```
WRITE +025
25
```

Each negative integer literal you use should consist of digits preceded by a Unary MINUS operator, for example:

-18

-235001

-12367444

2.3.1.2 Decimal Numeric Literals—A decimal numeric literal is a series of one or more digits and a single decimal point character (.) optionally preceded by a Unary MINUS character. The format for a decimal numeric literal is either of the following:

{-}*digit*(s).{*digit*(s)}

or:

{-}.*digit*(s)

Each positive decimal numeric literal greater than one should consist of a series of digits that is the integer part of the number followed by a decimal point and a series of digits that is the decimal fraction, for example:

4.332

6.819

18.05

Positive decimal numeric literals do not need a preceding Unary PLUS operator. DSM-11 discards the Unary PLUS operator in arithmetic operations, for example:

```
WRITE +63.6
63.6
```

The digits in the integer part of the number need not contain leading zeros. The digits in the fractional part of the number need not contain any trailing zeros. DSM-11 discards both leading and trailing zeros in arithmetic operations, for example:

```
WRITE 077.3450
77.345
```

Each positive decimal numeric literal less than one should consist of a decimal point followed by a series of one or more digits, for example:

.0915

.68

.321

Negative decimal numeric literals must contain a Unary MINUS operator (-) followed by the positive number that is the absolute value of the negative number, for example:

-4.332

-.0915

Negative decimal numeric literals also need not contain leading zeros in the integer portion or trailing zeros in the decimal portion. DSM-11 discards both leading and trailing zeros, for example:

```
WRITE -0.2500
-.25
```

2.3.1.3 Exponential Notation—DSM-11 recognizes both integer and decimal numeric literals in a range of plus or minus 10 to the power of plus or minus 26. You can enter very large or very small decimal numeric literals within this range by using *exponential notation*. The format for exponential notation is:

{-}*mantissa*E{-}*exponent*

where:

{ - }	is the optional Unary MINUS operator used with negative numbers
mantissa	is the decimal or integer number to be exponentiated
E	represents times 10 to the power of
{ - }	is the optional Unary MINUS operator used with a negative exponent
exponent	is the integer exponent (the power of 10).

For example, to enter the number 10, type

1E1

To enter the number 280, type:

2.8E2

To enter the number -.0481, type:

4.81E-2

DSM-11 can interpret numeric literals in exponential notation; but it does not use exponential notation in mathematical calculations. DSM-11 considers all numbers to be variable-length strings. Whenever you display mathematical calculations, DSM-11 displays the results as a string, for example:

```
WRITE 10E2
1000

WRITE 1.2345678901234E15
1234567890123400
```

In all mathematical operations, DSM-11 returns at least 15 significant digits. In additive operations (addition and multiplication), however, DSM-11 can return a greater precision.

2.3.2 String Literals

String literals are sets of zero or more of the 95 ASCII graphic characters (ASCII values 32 through 126) enclosed in double quotation marks. The only limitation on the length of a string literal is the maximum length of a DSM-11 line. That is, the string literal, its delimiting quotation marks, and all other characters on the line (such as commands and spaces) must not exceed 255 characters.

In the following statement "PATIENT'S NAME ?" is a string literal:

```
READ !,"PATIENT'S NAME ?",NAM
```

The value of a string literal is a function of its spelling. Every character, including the space character, counts. For example, the following strings are not the same:

"DIGITAL"

" DIGITAL"

The absence of characters counts as well. The string literal denoting an empty, or null, string is:

""

The enclosing quotation marks are delimiters of string literals. You must use them to specify a string literal but DSM-11 does not consider them part of the value of the literal and does not display them when it executes the line on which you entered them.

```
WRITE "THIS IS A STRING LITERAL"
THIS IS A STRING LITERAL
```

To include quotation marks in the string literal, type two additional pairs of quotation marks within the string. Whenever DSM-11 finds a pair of quotation marks inside the delimiting quotes, it interprets the pair as a single set of quotation marks.

```
WRITE """THIS IS A STRING LITERAL"""
"THIS IS A STRING LITERAL"

WRITE "THIS IS A ""STRING"" LITERAL"
THIS IS A "STRING" LITERAL
```

DSM-11 counts such pairs of embedded quotation marks as one character, not two, when it determines string length; but it does count both members of the pair when it determines the length of the line of which the string is a part.

2.4 Variables

A *variable* is a symbolic name that references a storage location. The variable uses the value of that storage location as its own value. This value can change during the execution of a routine. For example, as a result of the following statement, the variable A has the value 142.432:

```
SET A=44.51*3.2
```

The value for the variable, NAM, in the following statement is determined in a different way:

```
READ !,"PATIENT'S NAME ?",NAM
```

The variable has a value that depends on user response to the string-literal prompt:

```
PATIENT'S NAME ?
```

DSM-11 always uses the most recently assigned value of a variable when performing calculations. That value remains the same until DSM-11 encounters a statement that assigns a new value to the variable.

If after entering the statement:

```
SET B=3*4
```

you enter the statement:

```
SET B=4*6
```

the variable B has the value 24.

The data that variables reference is all of one type. DSM-11 stores all data as variable-length strings. (A string is a series of characters entered and stored as a unit.) Each string can consist of between zero and 255 characters.

Nevertheless, depending on the operation it is performing, DSM-11 can interpret a data string as either a numeric or an alphanumeric value. (See Section 2.7 for more information on the interpretation of data strings.)

Data strings entered as numeric values are like numeric literals. DSM-11 places the same restrictions on such data as it does on numeric literals. (See Section 2.3.1 for more information on numeric literals.)

Alphanumeric data are like string literals. They are data entered into the system whose only value is a function of their spelling. Unlike string literals, alphanumeric data can include any of the 128 characters from the ASCII character set.

DSM-11 has three types of variables:

- Local variables
- Global variables
- Special variables

Local and global variables can be defined and changed directly by application routines. Special variables are defined by the DSM-11 system and, with certain exceptions discussed in Chapter 7, cannot be changed directly by a user.

The following sections discuss the three types of variables in detail.

2.4.1 Local Variables

Local variables are names of storage for data values maintained in memory. They are temporary (existing only until you delete them or log off DSM-11) and accessible only from your partition.

You can define a local variable by referencing its name in a SET statement or a READ statement. The first character in a local variable name must be an alphabetic character or a percent (%) character. By convention, the percent character is used only in library routines. (See the *DSM-11 User's Guide* for more information on library routines.)

The other characters in a local variable name can be any combination of alphabetic or digit characters. For example, the following are valid local variable names:

NAM

sum

A

%AA

C2345678

DSM-11 uses only the first eight characters you enter as the variable name. If you enter as a variable name:

C23456789

DSM-11 assigns the variable the name:

C2345678

Therefore, you should not use variable names longer than eight characters that DSM-11 can shorten to the same spelling.

Local variables can be either simple or subscripted. A simple local variable contains a single datum. You create and reference it only by the variable name. Subscripted local variables have the value of data grouped into ordered sets or arrays. You can create and reference one by using the variable name and one or more subscripts.

The subscript is a string enclosed in parentheses immediately following the local variable name. It uniquely identifies a specific element in a local array.

Each subscript is a string that can consist of up to 63 characters. The characters can be any of the characters in the ASCII set except the null character (ASCII decimal value 0). Thus, subscripts can be:

- Integer Literals (positive or negative)
- Decimal Numeric Literals (positive or negative)
- String Literals

The following variables all have valid subscripts:

X(-1)

X(1.234)

TEST("FIRST","SECOND","THIRD")

Z("#")

A local array can be one dimensional or multidimensional. That is, an array element can have one or more subscripts. If you use more than one subscript, separate the subscripts by commas. Leave no spaces between the subscripts and the delimiting commas.

DSM-11 has no limit on the depth of subscripting you use. However, the full reference (variable name and all subscripts) cannot exceed 120 characters. For example, the node TEST(X) is invalid if X is a 117 character string.

Both simple and subscripted local variables can share the same name. Thus, the array ABC and the simple local variable ABC can exist at the same time.

2.4.2 Global Variables

Global variables are names of storage locations for data maintained on disk. Global variables are semipermanent, existing until they are specifically deleted. Because they are stored on disk, global variables are accessible to any authorized user.

You can define a global variable by referencing its name in a SET statement. The first character in a global variable name must be a circumflex (^) character. The second character must be either an alphabetic character or a percent (%) character. By convention, the percent character is used for globals in the library routines. (See the *DSM-11 User's Guide* for more information on library routines.)

The other characters can be any combination of alphabetic or digit characters. Thus, the following names are legal global variable names:

^NAM

^sum

^A

^%AA

^C2345678

DSM-11 uses only the first nine characters you enter as the variable name. For example, if you enter as a variable name:

^C23456789

DSM-11 assigns the variable the name:

^C2345678

Thus, you should not use variable names longer than nine characters that DSM-11 can shorten to the same spelling.

Global variables can be either simple or subscripted. A simple global variable has the value of a single datum. You create and reference it only by the variable name.

Subscripted global variables have the value of a node (element) in an array. (Such arrays are called globals.) You can create and reference a subscripted variable by using the variable name and a subscript.

The subscript is a string enclosed in parentheses immediately after the global variable name. It uniquely identifies a specific node or element in a global array.

Each subscript is a string that can consist of up to 63 ASCII characters. The characters can be any of the characters in the ASCII set except the null character (decimal ASCII 0). Thus, subscripts can be:

- Integer Literals (positive or negative)
- Decimal Numeric Literals (positive or negative)
- String Literals

The following global variables all have valid subscripts:

^X(-1,-2)

^X(3,7)

^X(1.234)

^X("FIRST","SECOND","THIRD")

A global array can be one dimensional or multidimensional. That is, an array node can have one or more subscripts. If you use more than one subscript, separate the subscripts by commas. Leave no spaces between the subscripts and the delimiting commas.

DSM-11 has no limit on the depth of subscripting you use. However, the full reference (variable name and all subscripts) cannot exceed 120 characters. For example, the node ^TEST(X) is invalid if X is a 117 character string.

2.4.3 Extended Global References

DSM-11 also provides you with the ability to extend global references to specify global variables in another UCI (User Class Identifier) or from another volume set or system. The syntax for such a reference is:

^ [*UCI*{,*SYS*}]*name*(*subscripts*)

where:

UCI	is the UCI in which the global is referenced
SYS	is the system or volume set on which the global resides

A volume set is a data base residing on one or more disks. There be more than one volume set on one system. If the UCI that you reference is undefined, you get a NOUCI error. If the system or volume set is undefined, you get a NOSYS error. See the *DSM-11 User's Guide* for more information on extended global references, volume sets, and intersystem communications.

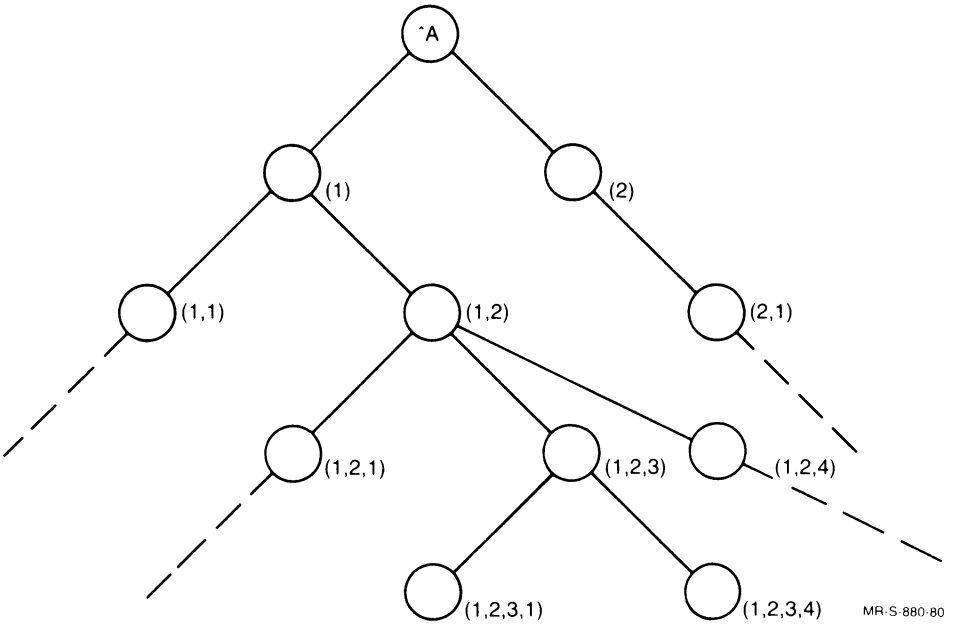
2.4.4 Array Structure

All local and global arrays are sparse arrays. DSM-11 does not require that you preallocate space for all possible nodes. Instead, it dynamically adds nodes to the array as you define them and deletes nodes when you delete them.

Both local and global arrays have a similar logical structure. This section describes global array structure.

Figure 2-1 shows the logical structure of a simple global array.

Figure 2-1: Global Array Structure



As Figure 2-1 shows, a global array is logically tree structured. The top, unsubscripted level is called the *root*. The lower, subscripted levels are called *branches*.

(By convention, global arrays are drawn as inverted trees with the root at the top and the branches beneath.)

All global-array nodes fall into levels depending on their depth of subscripting. In the global array tree shown in Figure 2-1:

^A	is on the root level (name level)
^A(1) ^A(2)	are on the first subscripting level
^A(1,1) ^A(1,2) ^A(2,1)	are on the second subscripting level
^A(1,2,1) ^A(1,2,3) ^A(1,2,4)	are on the third subscripting level
^A(1,2,3,1) ^A(1,2,3,4)	are on the fourth subscripting level

A global array is like a family tree. Consider again the global ^A previously shown in Figure 2-1. For any node, all nodes on the path between that node and the root are called ancestors. The ancestor on the next higher subscripting level (towards the root) is called the parent. Thus, in the global array ^A, the following nodes are the ancestors of ^A(1,2,3,1):

^A
^A(1)
^A(1,2)
^A(1,2,3)

While the node ^A(1,2,3) is the parent of ^A(1,2,3,1).

All nodes on a given level that have the same parent are called *siblings*. Thus, in the global array ^A, the following nodes are siblings:

^A(1,2,1)
^A(1,2,3)
^A(1,2,4)

They all have the same parent, ^A(1,2).

All nodes on a lower level that you can reach from a given node are called *descendants* of that node. In the global array ^A, the following nodes are the descendants of ^A(1,2):

^A(1,2,3)
^A(1,2,3,1)
^A(1,2,3,4)

Physically, DSM-11 stores only those global nodes that you explicitly define and the root. If you do not assign a value to the root of a global, DSM-11 assigns the root a null value. If you do not assign a value to the root of a local array, DSM-11 does not assign the root a value.

Logically, DSM-11 also defines any ancestor nodes it needs to provide a direct path from the root to the defined node. For example, if you define a global node ^X(1,3,5) in a previously undefined global array:

```
SET ^X(1,3,5)=1
```

DSM-11 physically creates a directory and a pointer for ^X and defines:

^X = ""

^X(1,3,5) = "1"

DSM-11 also defines the following logical ancestors to maintain a direct path from the root to the defined node:

^X(1)

^X(1,3)

DSM-11 does not maintain such logical ancestors physically. However, you can examine them with the $DATA, $ORDER, and $ZSORT functions. For example, $DATA shows ^X(1) as having no value, but as having descendants.

Thus, all nodes in DSM-11 can have one of three states:

1. A node can have a value and no descendants. That is, the node is explicitly defined and on the lowest level of the global array tree.

2. A node can have no value but have descendants. That is, the node is a logical node that maintains a path between the root and the node's descendants.

3. A node can have a value and have descendants. That is, the node is a physical node you have defined that also contains a pointer to maintain a path between the root and the node's descendants.

DSM-11 returns all nodes in the collating sequence of their subscripts. The system supports two collating sequences:

- Numeric sequence
- ASCII sequence

In *numeric sequence*, DSM-11 sorts nodes in the following order:

1. DSM-11 sorts all nodes with canonic numbers as subscripts in the ascending numeric order of their subscripts. Negative canonic subscripts sort first, then a subscript of zero, and finally positive canonic subscripts.

   NOTE

   A *canonic number* is a number reduced to its simplest form. Such a number contains only the valid numeric characters and, optionally, a negative sign and/or single decimal point. It has no leading zeros before the digits to the left of the decimal point and no trailing zeros after the digits to the right of the decimal point. An integer number followed by a decimal point (and no other digits), is not a canonic number.

2. DSM-11 then sorts all nodes with subscripts co string literals and noncanonic numbers (such as 01 or -0.5) in the ascending order of the ASCII value of their subscript characters, evaluated from left to right.

For example, a local array containing the nodes (elements) A(-5), A(2) A(1), A(2.5), A("01"), A(3), A(20), A("B") numerically collates in the following order:

A(-5)
A(1)
A(2)
A(2.5)
A(3)
A(20)
A("01")
A("B")

In *ASCII sequence*, DSM-11 returns all nodes in the ascending order of the ASCII values of their subscript characters. For example, ASCII collates the previous local array A in the following order:

A(-5)
A("01")
A(1)
A(2)
A(2.5)
A(20)
A(3)
A("B")

See the *DSM-11 User's Guide* for more information on array structure and usage.

2.4.5 Naked References

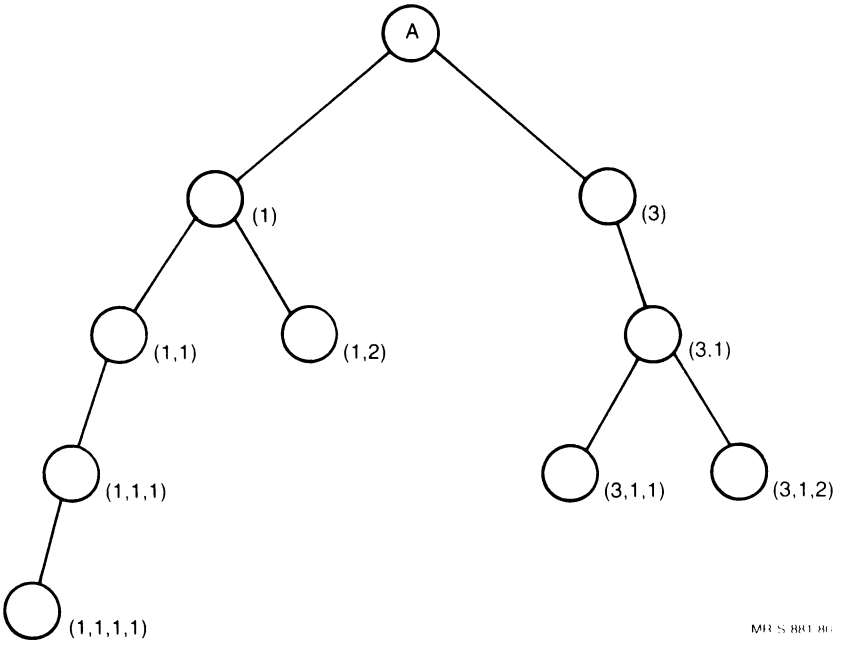
Naked references allow you to refer to a sibling or descendant of a previously referenced global node by using only that portion of the node's subscript that differs from the subscript of the node in the preceding reference. Naked references consist of the circumflex (^) character immediately followed by the unique portion of the subscript enclosed in parentheses.

Thus, if you make a full reference to the node ^ASG(1,2) and then want to reference the defined node ^ASG(1,3), you can do so with a naked reference ^(3).

DSM-11 maintains a pointer in your job's partition called the *naked indicator*. The naked indicator records the last named global reference and its level. Within a given global, the path of the naked indicator is always across a subscripting level of siblings and away from the root.

Figure 2-2 shows a simple global array ^A.

Figure 2-2: Global Array (Naked References)



If your last full reference was to ^A(1), the naked indicator is set to the first unsubscripted level (^A). You can now use naked references to:

1. ^A(3), the sibling of ^A(1) (as ^(3))

2. ^A(3,1), ^A(3,1,1), and ^A(3,1,2), the descendants of ^A(3) (as ^(3,1), ^(3,1,1), and ^(3,1,2))

3. ^A(1,1), the descendant of ^A(1) (as ^(1,1))

After you use a naked reference to ^A(1,1), DSM-11 sets the naked indicator to the first subscripting level. You can only use a naked reference to:

1. ^A(1,2), the sibling of ^A(1,1) (as ^(2))

2. ^A(1,1,1) and ^A(1,1,1,1), the descendants of ^A(1,1) (as ^(1,1) and ^(1,1,1))

To reference ^A(3) or any of its descendants, you must use a full global reference.

After you use a naked reference to ^A(1,1,1,1), the naked indicator is set to the third subscripting level. You cannot use a naked reference to any existing node; although you can use it to define any new nodes sibling or descendant to ^A(1,1,1,1). To "backtrack" in the array to a lower level node, you must use a full global reference.

The two types of full global references that leave the naked indicator undefined are:

1. A reference to the root of a global array

   For example, if the previous full global reference is S X=$D(^B), the following statement is undefined:

   ```
   SET ^(1)="LIST"
   ```

   DSM-11 returns a <NAKED> error.

2. A reference to a node of an undefined global array

See the description of $DATA, $NEXT, $ORDER, $ZNEXT, $ZORDER, and $ZSORT in Chapter 6 for more information.

2.4.6 Special Variables

Special variables are system defined and maintained variables that can provide you with information on your partition or on the system. During the course of processing, DSM-11 updates the values in these special variables. If you need to know the information they contain, you can access it by using the special variable as an argument to a DSM-11 command.

Each special variable is an alphabetic name preceded by a dollar sign. The name is a mnemonic for the information the special variable contains.

DSM-11 has two types of special variables:

- ANSI Standard special variables
- Extended special variables

ANSI Standard special variables are specified in the *ANSI MUMPS Language Standard* and follow standard usage. The standard reserves the characters $A through $Y as the first two characters of the standard special-variable names.

As specified in the *ANSI MUMPS Language Standard*, extended special variables are implementation-specific additions to the language. Extended special-variable names begin with the characters $Z.

When you use a special variable, you can enter either the full spelling or an abbreviation. If you are using an ANSI Standard special variable, you can abbreviate the spelling to the first two characters (the $ and the first letter of the variable name). For example, you can write the ANSI Standard special variable $TEST as one of the following:

$TEST

$T

When you use an extended special variable, you can abbreviate it to its first three characters (the $, the Z, and the first letter of the variable name). For example, you can write the extended special variable $ZTRAP as in either of the following ways:

$ZTRAP

$ZT

2.5 Unary Operators

Unary operators give an arithmetic or logical meaning to and are considered part of the expression atoms they precede. They include the unary arithmetic operators (Unary PLUS and Unary MINUS) and the Unary NOT operator.

Unary arithmetic operators indicate that the expression atom that they precede should be interpreted numerically. Whenever DSM-11 encounters a unary arithmetic operator, it uses the leftmost valid numeric characters in the value of the expression atom that follows as the numeric value of the expression atom.

```
WRITE +"56 DOLLARS AND 32 CENTS"
56
```

You can use the Unary NOT operator with both expression atoms or binary operators. If you use it with an expression atom, Unary NOT inverts the truth value of the expression atom. If you use it with a binary operator, Unary NOT inverts the meaning of the binary operator.

Because DSM-11 considers unary operators as part of their associated expression atom, it evaluates all unary operators in a right-to-left order before it evaluates any binary operators. For example:

```
WRITE 60--60
120
```

Table 2-1 lists the unary operators and their meanings.

Table 2-1: DSM-11 Unary Operators

Operator	Symbol	Example	Meaning
Unary NOT	'	' B	B is logically inverted.
Unary PLUS	+	+ B	B is interpreted as a numeric value.
Unary MINUS	-	- B	B is interpreted as a numeric value of the opposite sign.

2.6 Binary Operators

Binary operators indicate the type of value DSM-11 is to create from the expression atoms (or expressions) that they separate. In this situation, the expression atoms or expressions to the left and right of a binary operator are called the *operands* of that operator.

DSM-11 recognizes the following types of binary operators:

- Arithmetic operators
- Numeric relational operators
- String relational operators
- The string operator (Binary CONCATENATE)
- Logical operators

In addition, DSM-11 also recognizes certain other symbols, the *indirection* operator and *formatting characters*, that have the effect of operators.

2.6.1 Arithmetic Operators

Arithmetic operators evaluate their operands numerically and create a numeric (in one case, integer) result. Whenever DSM-11 encounters an arithmetic operator, it evaluates the character strings that are the values of the associated operands numerically.

DSM-11 uses the leftmost numeric characters in each string as the value of each operand. It then creates a value based on the numeric values of the operands and the operation indicated by the operator.

```
WRITE 25.5*4.6
117.3

WRITE "40 STARSHIPS"+"20 STARSHIPS"
60

SET A="22RAT",B="44CAT" WRITE A+B
66
```

If no numeric characters begin a string, DSM-11 gives the string a value of zero.

```
WRITE "40 STARSHIPS"+"TWENTY STARSHIPS"
40
```

Some DSM-11 functions, however, require an integer value as an argument. In these cases, DSM-11 first takes the numeric interpretation of the leftmost portion of the string and removes any decimal fraction. If the result is empty or contains only the Unary MINUS operator (indicating that the string has no whole integer value), DSM-11 gives the string a value of zero.

Table 2-2 lists the arithmetic operators and their meanings.

Table 2-2: DSM-11 Arithmetic Operators

Operator	Symbol	Example	Meaning
Binary ADD	+	A+B	Add B to A.
Binary SUBTRACT	-	A - B	Subtract B from A.
Binary MULTIPLY	*	A*B	Multiply A by B.
Binary DIVIDE	/	A/B	Divide A by B.
Binary INTEGER DIVIDE	\	A\B	Integer divide A by B.
Binary MODULO	#	A#B	A modulo B.

2.6.2 Numeric Relational Operators

Numeric relational operators are called *Boolean* operators. They evaluate two operands numerically and create a result that is a truth value of either one or zero.

If the numeric relationship between the operands is true, the result is a truth value of true (one). If the numeric relationship between the operands is false, the result is a value of false (zero).

You can also use the Unary NOT operator (') with a numeric relational operator to produce an inverted version of the operator. Table 2-3 lists the positive and inverse numeric relational operators and their meanings.

Table 2-3: Numeric Relational Operators

Operator	Symbol	Example	Meaning
Binary LESS THAN	<	A<B	A is less than B.
	'<	A'<B	A is not less than B. (A is greater than or equal to B.)
Binary GREATER THAN	>	A>B	A is not greater than B.
	'>	A'>B	(A is less than or equal to B.)

2.6.3 String Relational Operators

String relational operators are also Boolean operators; they compare their operands and create a truth value of either one or zero. They use the ASCII values of their operands to produce a result.

If the relationship between the two operands is true, the result is a truth value of true (one). If the relationship between the operands is false, the result is a truth value of false (zero).

You can also use the Unary NOT operator (') with the string relational operators to create an inverted version of the operator. Table 2-4 lists both the positive and inverted string relational operators and their meanings.

Table 2-4: String Relational Operators

Operator	Symbol	Example	Meaning
Binary EQUALS	=	A=B	String A and string B are equal.
	'=	A'=B	String A and string B are not equal.
Binary CONTAINS	[	A[B	String B is contained within string A.
	'[	A'[B	String B is not contained within string A.
Binary FOLLOWS	]	A]B	String A follows string B in ASCII collating sequence.
	']	A']B	String A does not follow string B in collating sequence.
Binary PATTERN MATCH	?	A?*patn*	String A is a string in the form specified by *patn*.
	'?	A'?*patn*	String A is not a string in the form specified by *patn*.

With the SET command, the equals character (=) is used in a way that is distinct from its use as a relational operator. With SET, the equals character is an assignment operator that has the meaning give the operand on the left the value of the operand on the right. (See the description of the SET command in PART 2 for more information.)

2.6.4 The String Operator

DSM-11 has one string operator, the Binary CONCATENATE operator (_). Binary CONCATENATE gives no special interpretation of its arguments. It creates a string that contains the characters of the right operand appended to the characters of the left operand.

```
WRITE "ROUND"_"TABLE"
ROUNDTABLE

WRITE 711_560
711560
```

2.6.5 Logical Operators

Logical operators produce a truth-valued result based on the truth values of their operands. When DSM-11 encounters a logical operator, it first interprets the operands numerically. If an operand evaluates to any value but zero, DSM-11 gives it a value of true (one). If an operand evaluates to zero, DSM-11 gives it a value of false (zero).

Then, DSM-11 evaluates the truth value of the expression by the truth values of the operands and the relationship between them described by the operator. If the relationship between the truth values of the operands is true, the result is true. If the relationship between the truth values of the operands is false, the result is false.

You can also use the Unary NOT operator (') with a logical operator to invert the meaning of the operator. Table 2-5 lists both the positive and inverted logical operators and their meanings.

Table 2-5: Logical Operators

Operator	Symbol	Example	Meaning
Binary AND	&	A&B	Both A and B are true.
	'&	A'&B '(A&B)	Either A or B or both are false.
Binary OR	!	A!B	Either A or B or both A and B are true.
	'!	A'!B '(A!B)	Both A and B are false.

Table 2-6 is a truth table. It describes the results of the previous logical operators on different combinations of truth-valued operands.

Table 2-6: Truth Table

A Value	B Value	A&B	A'&B	A!B	A'!B
0	0	0	1	0	1
1	0	0	1	1	0
0	1	0	1	1	0
1	1	1	0	1	0

2.6.6 Indirection Operator

The *indirection* operator allows you to use the value of an expression (expression atom) as an element in DSM-11 statements. The occurrence of indirection is represented by the indirection operator (@) followed by an expression atom.

Whenever DSM-11 encounters an occurrence of indirection, it replaces the occurrence with the value of the expression atom and uses that value in the statement. Thus, DSM-11 evaluates the statements:

```
SET X="A"
SET @X=40
```

as equivalent to:

```
SET A=40
```

Indirection is not a simple substitution of the string value of the expression atom for the occurrence of the expression atom. It is an evaluation and use of that value as part of a DSM-11 statement.

For example, the following statements substitute the value of X for the occurrence of X:

```
SET X="!!,""THIS IS NOT INDIRECTION""" W X

!!,"THIS IS NOT INDIRECTION"
```

The following statements use indirection to evaluate X:

```
SET X="!!,""THIS IS INDIRECTION""" W @X

THIS IS INDIRECTION
```

In the previous examples, the value of indirection must include the two additional sets of quotation marks. The exclamation points are formatting characters. (See Section 2.6.7 for more information on formatting characters.)

All indirection must evaluate to:

1. One or more command arguments
2. A name
3. A pattern (used with the Binary PATTERN MATCH operator)
4. A subscripted local or global variable name (used in partial indirection)

The following sections describe each form of indirection.

2.6.6.1 Argument Indirection—In argument indirection, the value of the indirection must be one or more command arguments. In the following example, DSM-11 sets C to the value of five and sets NAM(5,B+1) to the value of K.

```
S6  SET J="NAM(5,B+1)=K"
    SET C=5,@J
```

In the following example, DSM-11 uses indirection to transfer control to line A1 in the routine UPDATE.

```
J1  SET GOARG="A1^UPDATE"
    GO @GOARG
```

2.6.6.2 Name Indirection—In name indirection, the value of the indirection can be any DSM-11 *name*. DSM-11 defines as a name any language element that contains an uppercase alphabetic character or percent character followed by up to seven alphanumeric characters. Thus, you can use name indirection for:

- Variable names
- Line labels
- Routine names

When you use indirection to reference a named variable, the replacement value of the indirection must evaluate to a complete variable name, including subscripts. In the following example, DSM-11 sets the variable A to the value of four.

```
L2  SET X="A",@X=4
```

When you use indirection to reference a line label, the replacement value of the indirection must evaluate to any DSM-11 line label. In the following example, DSM-11 sets SEC to the value of the line label PART2 if ANS has the value of a null string. Later, DSM-11 passes control to the second line following the line labeled PART2.

```
J1  IF ANS="" SET SEC="PART2"


    GO @SEC+2
```

When you use indirection to reference a routine name, the replacement value of the indirection must evaluate to a syntactically valid DSM-11 routine name.

In the following example DSM-11 uses the value of a local variable to determine the routine to which it is to transfer control by indirection. DSM-11 uses the $SELECT function to determine the choice.

See the description of $SELECT in Chapter 6 for more information.

```
GET  READ !,"FUNCTION ?",A
     SET X=$SELECT(A=1:"TEST",A=2:"UPDATE",1:"EDIT")
     DO ^@X
```

2.6.6.3 Pattern Indirection—In pattern indirection, the value of the indirection must be a valid pattern. See the description of the binary PATTERN MATCH operator in Chapter 3 for a discussion of valid patterns.

In the following example, DSM-11 sets X to the pattern for one or more digits and A to the numeric literal 4. Then, it tests the truth value of A meeting the specified pattern with the binary PATTERN MATCH operator. Because the operand evaluates to one digit, the result is true (one).

```
SET X="1N.N",A=4 WRITE A?@X
1
```

You can nest indirection to give more flexibility to your routines, for example:

```
S1  SET VAR="A"
    SET TEST="$D(@VAR)=0"
    IF @TEST SET @VAR=""
```

Nested indirection is not recommended. It adds to system overhead and makes maintenance of application packages difficult.

2.6.6.4 Partial Indirection—The partial indirection of a subscripted variable allows you to construct a local or global variable reference by combining an indirect reference with a list of additional subscripts. The list of subscripts are not part of the indirect reference.

The syntax for partial indirection is different from the syntax for the other forms of indirection. Partial indirection requires a second indirection operator (@) to separate the indirected portion of the reference from a list of subscripts within parentheses. DSM-11 combines these two parts to form a single local or global reference that you can use to reference array nodes.

The general form of partial indirection is:

@expr atom@ (subscript1,... subscript n)

The first indirection operator precedes an expression that evaluates to a local or global variable name, defined or undefined. (Thus, the first "part" of subscript indirection is actually name indirection to a variable.)

The second indirection operator precedes a list of subscripts enclosed in parentheses; it immediately follows the expression atom without any intervening spaces.

To understand what partial indirection does, consider the following statements:

```
SET X="A(1)"
SET @X@(2,3)="HELLO THERE"
```

In this example, DSM-11 creates the variable A(1,2,3), and assigns it the value "HELLO THERE"

To perform the partial indirection, DSM-11 does two things. First, DSM-11 performs name indirection to a variable. In the preceding example, this action replaces the variable X with the string A(1), which is referred to as the indirect variable.

Second, DSM-11 creates the new array reference. To do this, it performs the following steps:

1. It deletes the right parenthesis from the subscript portion of the indirect variable.

2. It places a comma after the last character of the subscript of the indirect variable.

3. It concatenates all characters within the parentheses following the second subscript indirection operator to the indirect variable from step 2.

4. It puts a right parenthesis after the last character of the variable created as a result of step 3.

If the evaluation of partial indirection returns an unsubscripted variable in the name indirection phase, DSM-11 creates an array reference that consists of the variable name and all characters to the right of the second indirection operator (including parentheses), for example:

```
SET Y="A"
SET @Y@(1,2)="FIRST NODE"
WRITE A(1,2)
FIRST NODE
```

2.6.7 Formatting Characters

The *formatting characters* allow you to format output with the READ and WRITE commands. Table 2-7 lists the formatting characters and their meanings.

Table 2-7: Formatting Characters

Formatting Character	Meaning
#	Form feed
!	Carriage return/line feed (new-line operation)
?n	Horizontal tabulation --- positions the next character n spaces from the left margin

Formatting characters are separate READ or WRITE arguments. You must separate them from all preceding and following arguments by commas and no intervening spaces.

For example, both of the following statements are valid:

```
READ !,"NAME?",NAM,!,"AGE?",AGE,!

WRITE ?10,NAM,?21,AGE,!
```

You can use more than one format character or more than one type of formatting character in an argument. DSM-11 evaluates and executes each formatting character in left-to-right order. In the following statement, DSM-11 performs three new-line operations, tabulates to the 35th column, and prints the string "END OF REPORT".

```
WRITE !!!?35,"END OF REPORT"
```

2.6.7.1 The Form-Feed Character—The *form-feed character* (#) causes a top-of-form operation on your current device. DSM-11 recognizes the form-feed character only on hard-copy devices and spooling devices.

When you enter a form-feed character in a statement, DSM-11 writes a form feed and sets the $X and $Y special variables to zero before executing the argument to the right of the form-feed character.

Thus, when DSM-11 encounters the statement:

```
WRITE #,"CUSTOMER BILLING REPORT",!
```

It first writes a form feed to the current device before printing the string "CUSTOMER BILLING REPORT".

The special variable, $X, contains the current column position on your input/output device. The special variable, $Y, contains the number of lines used on the current input/output device since the last form feed. See the descriptions of $X and $Y in Chapter 7 for more information.

2.6.7.2 The Carriage-Return/Line-Feed Character—The *carriage-return/line-feed character* (!) causes a new line operation on the current device. When you enter a carriage-return/line-feed sequence character, DSM-11 starts a new line, sets $X to zero, adds one to $Y, and places any following text in the first column (column 0) of the new line.

Thus, when DSM-11 encounters the statement:

```
WRITE !,CUST(M),!,USE(M)
```

It ends the current line, sets $X to zero, adds one to $Y, and writes the data contained in the local variable CUST(M) at the left margin of a new line. Then, it ends that line, sets $X to zero, adds one to $Y, and writes the data contained in USE(M) at the left margin on the next line.

2.6.7.3 The Horizontal-Tabulation Character—The *horizontal-tabulation character* (?), when followed by an integer-valued expression (represented in Table 2-7 by n), creates an effect similar to tab to column n. That is, DSM-11 writes enough spaces to place any text that follows the horizontal-tabulation character at the nth column on the current line.

All horizontal tabulation is relative to the current value of $X. DSM-11 subtracts the current value of $X from the value specified in n and uses the remainder (n-$X) as the number of spaces to enter.

For example, if the current value of $X is three, then the following statement writes two spaces and the string HELLO.

```
WRITE ?5,"HELLO"
```

If the current value of $X is eight, the following statement writes two spaces and the value of ADDR(N).

```
WRITE ?10,ADDR(N)
```

DSM-11 cannot perform a negative tabulation. The interger you specify with the horizontal-tabulation character must be greater than or equal to the current value of $X. That is, you cannot specify an integer value of six or less if the current value of $X is seven.

Tabulation is relative to the absolute left margin. Each successive TAB formatting character you use must indicate the number of columns from the left margin, not from the last character formatted.

Thus, the following statement does not print ALPHA 10 columns from the left margin and BETA 10 columns from ALPHA:

```
WRITE ?10,"ALPHA",?10,"BETA"
```

Instead, the statement prints:

```
ALPHABETA
```

(In any line of text, if one horizontal-tabulation formatted string overlaps the starting column of another horizontal-tabulation formatted string, the second string starts at the next available column.)

To create the desired result, use the following statement:

```
WRITE !?10"ALPHA",?25,"BETA"
```

You can also create the same effect using $X. This allows you to leave a fixed number of spaces between the various strings independent of the actual length of the strings. Using $X, you can rewrite the previous statement as:

```
WRITE ?$X+10,"ALPHA",?$X+10,"BETA"
```

2.7 Expression Evaluation

DSM-11 evaluates all expressions in the following order:

1. DSM-11 evaluates and replaces the occurrences of indirection with the value of the indirection in left-to-right order.

2. DSM-11 evaluates all unary operators and applies them to the operand on their right, for example:

```
WRITE 27--5
32
```

   If you preceded an operand with multiple unary operators with an operand, DSM-11 applies the unary operators one by one, in right-to-left order, for example:

```
WRITE 27---5
22
```

   or:

```
WRITE 27----5
32
```

3. DSM-11 evaluates all binary operators in left-to-right order. All binary operators are on the same precedence level; that is, DSM-11 does not have a special evaluation order for binary operators.

Consider the following statement:

```
WRITE 44/4*2+6-1
```

DSM-11 evaluates the argument in four steps:

- 44/4 = 11
- 11*2 = 22
- 22 + 6 = 28
- 28-1 = 27

You can change the left-to-right order of binary-operator evaluation by enclosing expressions in parentheses. DSM-11 considers any expression in parentheses to be a single expression atom (operand). It evaluates all expressions in parentheses (in left-to-right order) before it evaluates the other binary operators.

Consider the following statement:

```
WRITE 44/(4*2)+(6-1)
```

DSM-11 evaluates this argument in four steps:

- 4*2 = 8
- 6-1 = 5
- 44/8 = 5.5
- 5.5 + 5 = 10.5

Consider also the following statement:

```
WRITE 'A&B
```

The result is true (one) only when A is false and B is true; but:

```
WRITE '(A&B)
```

creates a true result when:

- A and B are false.
- A is false and B is true.
- A is true and B is false.

You can nest parentheses to change the order of binary-operator evaluation. DSM-11 evaluates the expressions in the innermost sets of parentheses and creates a result. Then, it applies the results to the next outermost set of parentheses and so forth until it evaluates the whole expression.

Consider the following statement:

```
WRITE 44/(((4*2)+6)-1)
```

DSM-11 also evaluates this statement in four steps:

- 4*2 = 8
- 8 + 6 = 14
- 14-1 = 13
- 44/13 = 3.38461538461

All expressions are variable-length character strings; but, depending on the nature of the operands, the type of the operator, and the context of the

expression, DSM-11 can interpret expressions in different ways and create different results. For that reason, DSM-11 recognizes five types of expressions:

- String expressions
- Numeric expressions
- Truth-valued expressions
- Postconditional expressions
- Timeout expressions

2.7.1 String Expressions

String expressions are those in which DSM-11 gives no special interpretation to the operands. That is, DSM-11 treats the operands as variable-length character strings.

String expressions contain the Binary CONCATENATE operator or the binary string relational operators. When DSM-11 evaluates a string expression, it creates a value. For the string relational operators, the value is a truth value of false (zero) or true (one). For the Binary CONCATENATE operator, the value is a string containing the values of both operands concatenated.

```
WRITE !,"ROUND"_"TABLE"

ROUNDTABLE

WRITE !,123_678.56

123678.56
```

All operands you use in string expressions must have a defined value. If the operands do not have a defined value, you receive an error message when you execute the string expression.

2.7.2 Numeric Expressions

Numeric expressions are those that DSM-11 interprets numerically. Numeric expressions contain binary arithmetic operators.

When DSM-11 encounters a binary arithmetic operator, it uses any leading numeric characters (the digits, the decimal point, the unary operators, and the

letter E) as the value of the operand. If no such characters are present, it gives the operand a value of zero. Then DSM-11 creates a numeric value that is the result of the indicated mathematical operation on the numeric values of the operands.

2.7.3 Truth-Valued Expressions

Truth-valued expressions are those in which DSM-11 treats the operands as Boolean values. Truth-valued expressions contain Binary Relational or Binary Logical operators.

If the relationship between the operands shown in the expression is true, DSM-11 creates a truth value of true (one) as a result. If the relationship between the operands shown in the expression is false, DSM-11 creates a truth value of false (zero) as a result.

2.7.4 Postconditional Expressions

A *postconditional expression* is a special use of a truth-valued expression. It is a truth-valued expression appended to a command or an argument that makes the execution of that command or argument dependent on its truth value.

2.7.4.1 The Command Postconditional Expression—The command postconditional expression is composed of a truth-valued expression immediately following the command name and separated from the command name by a colon (:). Because command postconditional expressions are considered to be part of the command, they are separated from the command argument list (if any) by a space.

The general format for a postconditional used with a command is as follows:

*command:postcond*SParg list

where:

command	is any command except ELSE, FOR, or IF
postcond	is a truth-valued expression
arg list	is the argument list

You can use a postconditional expression with all DSM-11 com except ELSE, FOR, and IF. If you do not use a postconditional expression after a command, DSM-11 executes the command unconditionally, subject to external restrictions such as a preceding ELSE or IF.

If you do use a postconditional expression after a command, DSM-11 evaluates the postconditional before it executes the command or any of the command arguments. If the postconditional expression has a truth value of true (one) and if the arguments are true or have no postconditional expressions, DSM-11 executes the command and its arguments. If the postconditional expression has a truth value of false (zero), DSM-11 ignores the command and its arguments.

Because postconditional expressions do not affect the $TEST special variable, they are not equivalent to the IF command. However, you can use postconditional expressions in many situations where you would use IF.

For example, the following statement was written with the IF command:

```
IF J=4 SET REG=6
```

You can also write it with a postconditional expression:

```
SET:J=4 REG=6
```

2.7.4.2 The Argument Postconditional Expression—The argument postconditional expression is composed of a truth-valued expression immediately following an argument and separated from the argument by a colon. The general format for a postconditional expression with an argument is:

...argument:postcond...

where:

argument	is the argument you want to make conditional on the truth of the postconditional expression
postcond	is a truth-valued expression

You can use argument postconditional expressions with arguments of DO, GOTO, and XECUTE commands. If you do not use a postconditional expression, DSM-11 executes the argument unconditionally depending on external restrictions (such as a postconditional expression with the command).

If you do use a postconditional expression, DSM-11 evaluates the postconditional expression before it evaluates and executes the argument. If the postconditional expression is true, DSM-11 executes the argument. If the postconditional expression is false, DSM-11 skips the argument and executes any following arguments.

You can mix both command and argument postconditional expressions. For example, in the statement:

```
GOT DO:$DATA(A)#10 GT:A)0,LT:A(0,EQ:A=0
```

DSM-11 executes the DO command if A has a value. If the value is greater than zero, DSM-11 executes the code beginning at label GT. If the value is less than zero, DSM-11 executes the code beginning at label LT. If the value is zero, DSM-11 executes the code beginning at label EQ.

2.7.4.3 Naked References And Postconditional Expressions—If you use a postconditional with a statement containing a naked reference, the evaluation of the postconditional can have side affects even when the postconditional has a truth value of false. Consider the statement:

```
M1 SET:$DATA(^(1,2))=1 Y=Y+1
```

Even when the postconditional is false, DSM-11 increases the depth of the subscript level referenced in the naked indicator by one every time it evaluates the statement.

2.7.5 Timeout Expressions

A timeout expression, or *timeout*, is an numeric expression, preceded by a colon, that you can append to the argument of an OPEN, LOCK, READ, or JOB command. The value of the timeout specifies the number of seconds DSM-11 is to try to perform the operation specified by the command before it evaluates the next command. Real numbers (with a fractional part) are rounded to the nearest integer.

The timeout is an effective way of preventing an interruption of processing when the system must wait to execute an I/O statement. For example, when you issue an OPEN statement for an I/O device:

OPEN *device*

DSM-11 tries as long as necessary until it can open the device. If another user has opened and is using the device, the wait could be a long one.

If you use a timeout in the statement:

OPEN *device::timeout*

DSM-11 tries to open the device only for the number of seconds you specify in the timeout. (The OPEN command requires two colons with a timeout.)

The value of the timeout should be a nonnegative integer. If the value of the timeout is negative, DSM-11 gives the timeout a value of zero.

If the timeout is a zero, DSM-11 determines if it can perform the operation. If it can, it executes the command and sets $TEST to one. If it cannot, it sets $TEST to zero. In either case, execution proceeds without delay.

If the timeout is a positive integer, DSM-11 suspends execution and tries once each second to complete the specified operation. If DSM-11 cannot complete the operation in the interval specified in the timeout, it sets the special variable $TEST to zero and continues routine execution. If DSM-11 can complete the operation in that interval, it sets $TEST to one and continues execution.

(If DSM-11 can complete the operation before the timeout occurs, it continues execution and ignores any remaining time in the timeout.)

Thus, if you use a timeout in a routine, you should test to determine if the operation is successful, for example:

```
OPEN 3::5 E  WRITE "BUSY" G T1
```

Part 2: Language Reference

Chapter 3
DSM-11 Operators

This chapter describes the DSM-11 operators and provides examples of their use.

3.1 Introduction to DSM-11 Operators

DSM-11 operators are symbolic characters that specify the operation to be performed and the type of value to be produced from their associated operand or operands. See Sections 2.5 and 2.6 for a general introduction to operators.

3.2 Operator Descriptions

The following are reference descriptions of each DSM-11 operator. Binary operators are described first, followed by unary operators.

Each description contains an explanation of the purpose, forms, and operation of the operator and several examples of operator usage. All command-line examples are shown as they would appear when entered from a terminal. All routine-line examples are shown as they would appear when listed on a line printer.

Binary ADD (+)

PURPOSE:

Binary ADD produces the sum of two numerically interpreted operands.

FORM:

operand + *operand*

EXPLANATION:

Binary ADD uses any leading, valid numeric characters (the digits 0 through 9, the decimal point, the Unary MINUS operator, the Unary PLUS operator, and the letter E) as the numeric values of the operands. Then, it produces a value that is the sum of the value of the operands.

```
W "8 APPLES"+"4 ORANGES"
12
```

If an operand has no leading numeric characters, Binary ADD gives it a value of zero.

```
W "8 APPLES"+"FOUR ORANGES"
8
```

COMMENTS:

Binary ADD produces at least 15 significant digits of precision in its result.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

EXAMPLES:

The following example performs string arithmetic on two operands that have leading digits.

```
W "4 motorcycles"+"5 unicycles"
9
```

The following example performs addition on two decimal, numeric literals.

```
W 2936.22+301.45
3237.67
```

Binary ADD (+) (Cont.)

The following example performs addition on two defined local variables.

```
S A=27.4,B=18.6 W A+B
46
```

The following example illustrates that leading zeros on a numerically evaluated operand do not affect the results the operator produces.

```
W "007"+10
17
```

Binary AND (&)

PURPOSE:

Binary AND tests whether both of its operands have a value of true.

FORM:

operand&*operand*

EXPLANATION:

Binary AND produces a true (one) only if both operands are true (that is, have nonzero values); otherwise, it produces a value of false (zero).

COMMENTS:

You can specify the Boolean operation of NOT AND (NAND) by using the Unary NOT operator with Binary AND. You can use either of the following forms;

operand'&*operand*

'(*operand*&*operand*)

Both forms are equivalent.

The negative AND reverses the truth value of Binary AND applied to both operands. It produces a true result only when either, but not both, operands are true or when both operands are false. It produces a false result only when both operands are true.

RELATED:

Expression Evaluation (Section 2.7)
Logical Operators (Section 2.6.5)
Truth-Valued Expressions (Section 2.7.3)

EXAMPLES:

The following example evaluates two nonzero-valued operands and produces a value of true (one).

```
S A=-4,B=-1 W A&B
1
```

Binary AND (&) (Cont.)

The following example evaluates one true and one false operand and produces a value of false (zero).

```
S A=1,B=0 W A&B
0
```

The following example evaluates two false operands with a negative AND. Thus, it produces a value of true (one).

```
S A=0,B=0 W A'&B
1
```

The following example evaluates one true and one false operand with a negative AND. Thus, it produces a value of true (one).

```
S A=0,B=1 W '(A&B)
1
```

Binary CONCATENATE (_)

PURPOSE:

Binary CONCATENATE produces a result that is a string composed of the right operand appended to the left operand.

FORM:

operand_operand

EXPLANATION:

Binary CONCATENATE gives the operands no special interpretation. It treats them as string values. If the concatenated string is longer than 255 characters, DSM-11 returns an error.

RELATED:

String Expressions (Section 2.7.1)
The String Operator (Section 2.6.4)

EXAMPLES:

The following example concatenates two string literals.

```
W "ROUND"_"TABLE"
ROUNDTABLE
```

The following example concatenates two numeric literals and a string literal.

```
W 609_"-"_24
609-24
```

The following example concatenates two local variables, NAME and SUBS.

```
S NAME="^AA",SUBS="(1,2,3)",GLOBAL=NAME_SUBS
W GLOBAL
^AA(1,2,3)
```

The following example concatenates two string literals and a null string.

```
S A="ABC"_""_"DEF" W A
ABCDEF
```

Thus, a null string has no effect on the length of a string. (You can concatenate an infinite number of null strings to a string.)

Binary CONTAINS ([)

PURPOSE:

Binary CONTAINS tests whether the sequence of characters in the right operand is contained in the sequence of characters in the left operand.

FORM:

operand A[*operand B*

where:

operand A is the operand to be considered as containing operand B

operand B is the operand to be considered as being contained in operand A

EXPLANATION:

Binary CONTAINS produces a result of true (one) if *operand A* contains the character string represented by *operand B* . Binary CONTAINS produces a result of false (zero) if *operand A* does not contain the character string represented by *operand A*.

To produce a true result, the characters in *operand B* must be in the same order as the characters in *operand A*. If *operand B* is a null string, the result is always true.

COMMENTS:

You can produce a negative CONTAINS (DOES NOT CONTAIN) by using the Unary NOT operator with Binary CONTAINS. You can write a DOES NOT CONTAIN in two ways:

operand A'[*operand B*

'(*operand A*[*operand B*)

Both forms are equivalent.

DOES NOT CONTAIN reverses the truth value of Binary CONTAINS applied to both operands. DOES NOT CONTAIN produces a result of true if *operand A* does not contain the character string represented by *operand B*. It produces a result of false if *operand A* does contain the character string represented by *operand B*.

Binary CONTAINS ([) (Cont.)

RELATED:

String Expressions (Section 2.7.1)
String Relational Operators (Section 2.6.3)

EXAMPLES:

The following example tests whether L contains S. Because L does contain S, the result is true.

```
S S="STEAM",L="STEAM LOCOMOTIVE"
W L[S
1
```

The following example tests whether the string represented by S is contained in P. Because the character sequence in the strings is different (two spaces in P and one space in S), the result is false.

```
S S="STAND UP",P="STAND  UP"
W P[S
0
```

The following example shows how you can use the DOES NOT CONTAIN to determine if one string is not contained in another string.

```
W "ABC"'["123"
1
```

The following example shows that the two operands can be equal in value and still produce a true result.

```
W "123"["123"
1
```

Binary DIVIDE (/)

PURPOSE:

Binary DIVIDE produces a quotient that is the result of dividing two numerically interpreted operands.

FORM:

operand A/operand B

where:

operand A is the dividend

operand B is the divisor

EXPLANATION:

Binary DIVIDE uses any leading, valid numeric characters (the digits 0 through 9, the decimal point, the Unary MINUS operator, the Unary PLUS operator, and the letter E) as the numeric value of the operand. Then, it produces a quotient that is the result of dividing *operand A* by *operand B*.

```
W "8 APPLES"/"4 ORANGES"
2
```

If an operand has no leading numeric characters, Binary DIVIDE gives it a value of zero.

```
W "8 APPLES"/"FOUR ORANGES"
```

The result of this operation is invalid. Dividing a number by zero causes an error.

COMMENTS:

Binary DIVIDE produces at most 31 significant digits of precision in its result. The number of signifigant digits is a parameter chosen during system generation; see the *DSM-11 User's Guide* for more information.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

Binary DIVIDE (/) (Cont.)

EXAMPLES:

The following example divides two integer numeric literals.

```
W 355/113
3.14159292035398
```

The following example performs division on operands with leading digits.

```
W "12 BATTLESHIPS"/"3 AIRCRAFT CARRIERS"
4
```

The following example performs division on two literals in E format. DSM-11 displays the result in numeric format.

```
W 10E4/5E2
200
```

Binary EQUALS (=)

PURPOSE:

Binary EQUALS compares two string-interpreted operands for equality.

FORM:

operand = operand

EXPLANATION:

Binary EQUALS normally tests for string equality. If the two operands tested are identical strings, Binary EQUALS produces a result of true (one); otherwise, it produces a result of false (zero).

To produce a true result, the character sequence in both operands must be identical. There can be no intervening characters (including spaces).

Binary EQUALS does not imply any numeric interpretation of either operand. Binary EQUALS tests string identity.

For example, the following statement produces a value of one:

```
W "SEVEN"="SEVEN"
```

But the next statement produces a value of zero:

```
W "007"="7"
```

The two operands in the second case are numerically identical, but their string values are different.

You can use Binary EQUALS to test for numeric equality if both operands have numeric values. Thus the following statement produces a value of one:

```
W 007=7
```

If the operands are not automatically converted to numeric values (as in the process of evaluating numeric literals), you can force the conversion by using the Unary PLUS operator. Thus, the following statement also produces a value of one:

```
W +"007"=+"7"
```

Binary EQUALS (=) (Cont.)

COMMENTS:

Keep the following points in mind when you use the Binary EQUALS operator.

1. You can specify a NOT EQUALS operation by using the Unary NOT operator with Binary EQUALS. You can express the NOT EQUALS operation in two ways:

   operand' = operand

   '(operand = operand)

   NOT EQUALS reverses the truth value of the EQUALS operator applied to both operands. If the two operands are not identical, NOT EQUAL produces a result of true. If the two operands are identical, it produces a result of false.

2. When used with the SET command, the equals character becomes an assignment operator that assigns the value of the operand on the right to the operand on the left, for example:

   ```
   S A="JOHN JONES"
   ```

   assigns the variable A the value "JOHN JONES".

RELATED:

Expression Evaluation (Section 2.7)
String Expressions (Section 2.7.1)
String Relational Operators (Section 2.6.3)

EXAMPLES:

The following example prompts for operator input. If the operator types a RETURN (equivalent to a null string), the routine terminates.

```
QU  R !,"ANY CHANGES?",ANS
    Q:ANS=""
```

The following example illustrates two uses of the equals character. First, the example uses the equals character as an assignment operator with the SET command to give two local variables the value of two strings. Second, it tests the identity of the strings with the Binary EQUALS operator. Because the strings are not identical, the result is false.

```
S A="A56BC",B="ABC" W A=B
0
```

Binary EQUALS (=) (Cont.)

The following example reads an answer, sets the variable TRUE to the value of the expression ANS = YES, and writes the result. If the contents of ANS is YES, then the expression to which TRUE is set is true (one). If the contents of ANS is any other value, the expression to which TRUE is set is false (zero).

```
B1  R !,ANS S TRUE=ANS="YES"
    W !,TRUE
```

The following statement does not set both A and B to 7:

```
S A=B=7
```

This statement sets A equal to true (one) if the value of B is 7; A is set to false (zero) if B has some other value.

Binary FOLLOWS (])

PURPOSE:

Binary FOLLOWS tests whether the characters in operand A come after the characters in operand B in ASCII collating sequence.

FORM:

operand A]*operand B*

where:

operand A is the operand that the test considers follows operand B

operand B is the operand that the test considers operand A follows

EXPLANATION:

Binary FOLLOWS compares the ASCII characters in both operands, starting with the leftmost character. It stops the comparison if it finds a character in *operand A* that is different from the corresponding character in *operand B*.

If Binary FOLLOWS finds that the first unique character in *operand A* has a higher ASCII value than the corresponding character in *operand B* (that is, if it comes after the corresponding character in *operand B* in ASCII collating sequence), it produces a result of true (one). If Binary FOLLOWS finds all characters in both operands are identical or if it finds that the first unique character in *operand A* has a lower ASCII value than the corresponding character in *operand B*, it produces a result of false (zero).

COMMENTS:

You can produce a NOT FOLLOWS operation by using the Unary NOT operator with Binary FOLLOWS. The NOT FOLLOWS operation has two forms:

operand A']*operand B*

'(*operand A*]*operand B*)

Both forms are functionally identical.

NOT FOLLOWS reverses the truth value of Binary FOLLOWS applied to both operands. NOT FOLLOWS compares the characters in both operands. It stops the comparison if it finds a character in *operand A* that is different from the corresponding character in *operand B*.

Binary FOLLOWS (]) (Cont.)

If all characters in the operands are identical, or if the first unique character in *operand A* has a lower ASCII value than a corresponding character in *operand B*, NOT FOLLOWS produces a result of true. If the first unique character in *operand A* is a higher ASCII value than the corresponding character in *operand B* (that is, if it follows the character in *operand B*), NOT FOLLOWS produces a result of false.

RELATED:

ASCII Character Set (Appendix A)
Expression Evaluation (Section 2.7)
String Expressions (Section 2.7.1)
String Relational Operators (Section 2.6.3)

EXAMPLES:

The following example tests to determine if the string LAMPOON follows the string LAMP in ASCII collating order. The result is true.

```
W "LAMPOON"]"LAMP"
1
```

The following example tests the collating order of numeric literals. Because 3 in 123 follows 2 in 122, the result is true.

```
W 123]122
1
```

The following example also tests numeric literals. Because the numeric literal 123 collates before the numeric literal 2, the result is false.

```
W 123]2
0
```

The following example tests to determine if the string CDE follows string ABC in ASCII collating order. Because C in CDE follows A in ABC, the result is true.

```
W "CDE"]"ABC"
1
```

The following example tests to determine if string CDE does not follow string ABC. Because C in CDE does follow A in ABC, the result is false.

```
W "CDE"']"ABC"
0
```

Binary FOLLOWS (]) (Cont.)

The following example tests if the string in B follows the string in A. Because BO follows BL in ASCII collating sequence, the result is true.

```
S A="BLUE",B="BOY"
W B]A
1
```

Binary GREATER THAN (>)

PURPOSE:

Binary GREATER THAN tests whether *operand A* is algebraically greater than *operand B*.

FORM:

operand A > *operand B*

where:

operand A is the operand considered the larger

operand B is the operand considered the smaller

EXPLANATION:

Binary GREATER THAN evaluates the two operands numerically. If *operand A* has a greater value than *operand B*, it produces a result of true (one). If *operand A* has an equal or lesser value, Binary GREATER THAN produces a result of false (zero).

COMMENTS:

You can produce a NOT GREATER THAN operation by using the Unary NOT operator with Binary GREATER THAN. The NOT GREATER THAN operation has two forms:

operand A' > *operand B*

'(*operand A* > *operand B*)

NOT GREATER THAN reverses the truth value of Binary GREATER THAN applied to both operands. Thus, you can use it to specify less than or equal to.

NOT GREATER THAN produces a true result when:

- *operand A* is less than *operand B*
- *operand A* is equal to *operand B*

NOT GREATER THAN produces a false result when operand A) is greater than *operand B*.

Binary GREATER THAN (>) (Cont.)

RELATED:

Expression Evaluation (Section 2.7)
Numeric Relational Operators (Section 2.6.2)
Truth-Valued Expressions (Section 2.7.3)

EXAMPLES:

The following example tests two numeric literals.

```
W 12>15
0
```

The following example tests two variables with the NOT GREATER THAN operator. Because both variables have an identical numerical value, the result is true.

```
S A="55",B="55" W A'>B
1
```

The following example tests two alphanumeric strings. Because neither string has leading numeric digits, both evaluate to zero. Thus the two strings are equal in numeric value and produce a true result only with the NOT GREATER THAN operator.

```
S A="ABC",B="BCD"
W A>B
0
W B>A
0
W A'>B
1
W B'>A
1
```

Binary INCLUSIVE OR (!)

PURPOSE:

Binary INCLUSIVE OR tests whether one or both of its operands have a value of true.

FORM:

operand!*operand*

EXPLANATION:

If either operand has the value of true or if both operands have the value of true, Binary INCLUSIVE OR produces a result of true (one). If both operands are false, it produces a result of false (zero).

COMMENTS:

You can produce a NOT OR (or NOR) operation by using Unary NOT with INCLUSIVE OR. The NOT OR operation has two forms:

operand'!*operand*

'(*operand*!*operand*)

Both forms are functionally equivalent.

The NOT OR operation reverses the truth value of Binary OR applied to both operands. If both operands are false, the NOT OR operation produces a result of true. If either operand is true or if both operands are true, the NOT OR operation produces a result of false.

RELATED:

Expression Evaluation (Section 2.7)
Logical Operators (Section 2.6.5)
Truth-Valued Expressions (Section 2.7.3)

EXAMPLES:

The following example evaluates two true operands, applies the OR to them, and produces a true result.

```
S A=1,B=1 W A!B
1
```

Binary INCLUSIVE OR (!) (Cont.)

The following example evaluates one true and one false operand and produces a true result.

```
S A=1,B=0 W A!B
1
```

The following example evaluates two false operands and produces a false result.

```
S A=0,B=0 W A!B
0
```

This NOT OR example evaluates two false operands and produces a true result.

```
S A=0,B=0 W A'!B
1
```

This NOT OR example evaluates one true and one false operand and produces a false result.

```
S A=1,B=0 W A'!B
0
```

Binary INTEGER DIVIDE(\)

PURPOSE:

Binary INTEGER DIVIDE produces the integer result of the division of *operand A* by *operand B* .

FORM:

operand A *operand B*

where:

operand A is the dividend

operand B is the divisor

EXPLANATION:

Binary INTEGER DIVIDE uses any leading, valid numeric characters (the digits 0 through 9, the Unary MINUS operator, the Unary PLUS operator, the decimal point, and the letter E) as the numeric values of the operands. Then, it returns a result that is the integer portion of the quotient produced by dividing *operand A* by *operand B*. (It does not return a remainder.)

```
W "8 APPLES"\"3.8 ORANGES"
2
```

If an operand has no leading numeric characters, Binary INTEGER DIVIDE gives it a value of zero. If you attempt integer division with a zero-valued divisor, you receive an error.

COMMENTS:

Binary INTEGER DIVIDE produces at most 31 significant digits of precision in its results. The number of significant digits is a parameter chosen during system generation; see the *DSM-11 User's Guide* for more information.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

Binary INTEGER DIVIDE(\) (Cont.)

EXAMPLES:

The following example performs integer division on two integer-literal operands.

```
W 355\113
3
```

The following example performs integer division on two real numeric operands.

```
W 163.83\128.8
1
```

The following example uses integer division to extract the integer part of a real number. The fractional part is truncated. DSM-11 performs no rounding-up operation.

```
S X=163.83
W X\1
163
```

The following example does perform the rounding-up operation.

```
S X=163.83
W X+.5\1
164
```

Binary LESS THAN (<)

PURPOSE:

Binary LESS THAN tests whether *operand A* is algebraically less than *operand B*.

FORM:

operand A < *operand B*

where:

operand A is the operand considered the smaller

operand B is the operand considered the larger

EXPLANATION:

If *operand A* has a lesser value than *operand B*, Binary LESS THAN produces a result of true. If *operand A* has an equal or greater value than *operand B*, Binary LESS THAN produces a result of false.

COMMENTS:

You can produce a NOT LESS THAN operation by using the Unary NOT operator with Binary LESS THAN. You can express this operation in two ways:

operand A' < *operand B*

'(*operand A* < *operand B*)

Both forms are functionally identical.

NOT LESS THAN reverses the truth value of Binary LESS THAN applied to both operands. It produces a true result when *operand A* is greater than *operand B* or when *operand A* is equal to *operand B*. It produces a false result when *operand A* is less than *operand B*.

Thus, you can use the NOT LESS THAN operation to specify "greater than or equal to."

RELATED:

Expression Evaluation (Section 2.7)
Numeric Relational Operators (Section 2.6.2)
Truth-Valued Expressions (Section 2.7.3)

Binary LESS THAN (<) (Cont.)

EXAMPLES:

The following example evaluates two numeric literals. Because nine is greater than six, the result is false.

```
W 9<6
0
```

The following example evaluates a string literal and a numeric literal. Because the numeric interpretation of the string is 22, the result is true.

```
W "22A4"<100
1
```

The following example evaluates two string literals. Because the strings have no leading digits, they both evaluate to zero. Thus, the result is true.

```
W "A"'<"B"
1
```

The following example shows the result of using relational operators in a series. All DSM-11 expressions with binary operators are evaluated left to right. The first operation is 2<X, the result of which is 0. The second operation is 0<10, the result of which is 1.

```
S X=0
W 2<X<10
1
```

Binary MODULO (#)

PURPOSE:

Binary MODULO produces a value that is *operand A* modulo *operand B*.

FORM:

operand A#operand B

where:

operand A is the value on which the modulo operation is to be performed

operand B is the modulus

EXPLANATION:

Binary MODULO uses any leading, valid numeric characters (the digits 0 through 9, the decimal point, the Unary MINUS operator, the Unary PLUS operator, and the letter E) as the numeric value of the operands. If an operand has no leading numeric characters, Binary MODULO gives it a value of zero.

DSM-11 defines modulo operation only for nonzero values of *operand B*. It defines *operand A #operand B* as:

*operand A -(operand B *floor)(A/B)*

where:

floor(A/B) is the largest integer less than or equal to *operand A/operand B*

For negative numbers, that integer is the largest absolute integer, that is, -6.2 becomes -7, not -6.

For example:

floor (1.2) = 1

floor (0) = 0

floor (-4.2) = -5

Binary MODULO (#) (Cont.)

COMMENTS:

When both operands are positive, the modulo operation produces the remainder of *operand A* /*operand B*. When one or both operands are not positive, the results are more complex.

The expression "is identical modulo" can help in understanding the process. Two numbers are identical modulo B if their difference is a multiple of B. In general, there are an infinite series of numbers identical to a specific number modulo B.

The formula that defines the modulo operation implies that:

- If B>0, then A#B produces the least positive number identical to A#B.
- If B<0, then A#B produces the greatest negative number identical to A#B.
- If B=0, then A#B is undefined and produces an error.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

EXAMPLES:

The following examples illustrate the modulo operation with two positive operands. In such cases, the modulo operation produces a value equivalent to the remainder after division of *operand A* by *operand B*.

```
W 37#10
7

W 24#4
0

W 12.5#3.2
2.9
```

Binary MODULO (#) (Cont.)

The following examples illustrate the effect of the modulo operation on two operands preceded with Unary MINUS operators. In such cases, the modulo operation is equivalent to:

-(*operand A#operand B*).

```
W -37#-10
-7

W -24#-4
0
```

The following example shows the effect of a Unary MINUS on *operand A*. In these cases, DSM-11 gives the expression a value of:

1. 0, if *operand A#operand B* has a value of 0.

2. *operand B*-(*operand A#operand B*), if *operand A#operand B* has a value other than 0.

```
W -37#10
3
```

These examples show the effect of a Unary MINUS on *operand B*. In these cases, the expression has the value -(-*operand A#operand B*).

```
W 37#-10
-3

W 24#-4
0
```

These examples show the effect of a zero-valued *operand A*. In such cases, the result is zero regardless of the sign of *operand B*. When *operand B* evaluates to zero, the operation is undefined and results in an error.

```
W "ALPHA"#10
0

W 0#-10
0
```

Binary MULTIPLY (*)

PURPOSE:

Binary MULTIPLY produces the product of multiplying two numerically interpreted operands.

FORM:

*operand*operand*

EXPLANATION:

Binary MULTIPLY uses any leading numeric characters (the digits 0 through 9, the Unary MINUS operator, the decimal point, and the letter E) as the numeric value of the operand. Then, it produces a product that is the result of multiplying the two operands.

```
W "8 APPLES"*"4 ORANGES"
32
```

If an operand has no leading numeric characters, Binary MULTIPLY gives it a value of zero.

```
W "8 APPLES"*"FOUR ORANGES"
0
```

COMMENTS:

Binary MULTIPLY produces at least 15 significant digits of precision in its result.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

EXAMPLES:

The following example multiplies two string operands with leading digits.

```
W "12 HORSES"*"4 COWS"
48
```

The following example multiplies one string literal and one numeric literal.

```
W "12.50"*.33
4.125
```

Binary MULTIPLY (*) (Cont.)

The following example multiplies the values in two local variables.

```
S B=25,A=4 W A*B
100
```

Binary PATTERN MATCH (?)

PURPOSE:

Binary PATTERN MATCH tests whether the pattern of characters in its left operand is correctly specified by the pattern that is its right operand.

FORM:

operand?*pattern*

where:

operand	is the operand whose characters you want to test for a pattern
pattern	is the pattern for which you want to test

EXPLANATION:

Binary PATTERN MATCH produces a result of true (one) if the pattern correctly specifies the pattern of characters in the operand. Binary PATTERN MATCH produces a result of false (zero) if the pattern does not correctly specify the pattern of characters in the operand.

The pattern characters you can use with PATTERN MATCH and their meanings are:

Character	Specifies
N	One of the ten numeric characters from 0 through 9
U	One of the 26 uppercase alphabetic characters from A through Z
L	One of the 26 lowercase alphabetic characters from a through z
A	One of the 26 uppercase or 26 lowercase alphabetic characters from A or a through Z or z
P	One of the 33 punctuation characters, including SP
C	One of the 33 control characters, including delete
E	One of any of the characters in the (entire) ASCII set

Binary PATTERN MATCH (?) (Cont.)

You can also use the following characters in patterns:

1. Integer literals

   Precede pattern characters with an integer literal that specifies the number of character matches necessary. For example, given the statement:

   ```
   S A="ALPHA CENTAURI"
   ```

   the statements:

   ```
   W A?5U1P8U
   ```

   (five uppercase alphabetics, one punctuation character, and eight alphabetics) or:

   ```
   W A?14E
   ```

   (fourteen of any of the characters in the ASCII set) both produce a true result.

   Place two or more code characters together to indicate that a match with any one of them is satisfactory. For example, the statement:

   ```
   W "CHARLESJONES"?12AN
   ```

   produces a true result because all the characters in the string are alphanumeric.

2. The period character

   You can use the period character (.) to do the following:

   a. Indicate that any number of occurrences constitute a match (including zero).

      To use the period this way, you include it in the pattern specification in place of integer values. For example, given the following statements, DSM-11 returns a true result because X does match a specification of any number of alphabetic or numeric characters.

      ```
      S X="A3B2A1" W X?.AN
      1
      ```

   b. Set lower bounds or upper bounds to the number of occurrences that constitute a match, or set a range of occurrences that constitute a match.

To use the period to set a lower bound, specify an integer followed by the period. For example, the following statement tests for an integer that consists of three or more digits:

```
W Y?3.N
```

To use the period to set an upper bound, specify the period followed by an integer. For example, the following statement tests for an uppercase alphabetic string that consists of zero to five characters.

```
W Y?.5U
```

To use the period to set a range of occurrences that match, specify an integer followed by a period followed by an integer. For example, the following statement tests for a lowercase alphabetic string that consists of two to seven characters.

```
W Y?2.7A
```

3. String values

Use a string value in place of part of a pattern to indicate that an exact character-for-character match is necessary. Thus, given the statement from the earlier example:

```
S A="ALPHA CENTAURI"
```

either of the statements below produces a true result:

```
W A?1"ALPHA"1P8U
```

(one string with the characters ALPHA, one punctuation character, and eight uppercase alphabetic characters)

```
W A?1"ALPHA"9E
```

(one string with the characters ALPHA and nine of any characters in the ASCII set)

You can precede any string with either an integer literal or the period character; but you cannot directly precede a string with any of the pattern characters. Thus, the following statement produces an error:

```
S A=",AB" W A?1P"AB"
```

Binary PATTERN MATCH (?) (Cont.)

COMMENTS:

When the period character precedes one or more pattern characters and the period is not preceded or followed by an integer, it applies to all pattern characters that follow through the occurrence of the next qualified period character (if any).

RELATED:

ASCII Character Set (Appendix A)
Expression Evaluation (Section 2.7)
Indirection Operator (Section 2.6.6)
Pattern Indirection (Section 2.6.6.3)
String Relational Operators (Section 2.6.3)
Truth-Valued Expressions (Section 2.7.3)

EXAMPLES:

The following example produces a true result. The string tested includes one numeric character, one punctuation character, and two numeric characters.

```
W "1,12"?1N1P2N
1
```

The following example produces a true result. The string tested includes four numeric characters, two uppercase alphabetics, one punctuation character, and one additional numeric character.

```
W "4227LZ-5"?3.N.U.2P1N
1
```

The following example produces a false result. The first character in the tested string is one of the 52 uppercase and lowercase alphabetics, but the test has no provision for the second character.

```
S A="CC" W A?1A
0
```

The following example produces a false result. The first two characters are alphanumeric, but the third character is punctuation.

```
W "6A-"?3AN
0
```

The following example produces a true result. The string "FIL" matches the first three characters and the 1E matches the last character.

```
W "FILL"?1"FIL"1E
1
```

Binary SUBTRACT (-)

PURPOSE:

Binary SUBTRACT produces the difference between two numerically interpreted operands.

FORM:

operand A-operand B

where:

operand A	is the minuend (the value from which *operand B* is to be subtracted)
operand B	is the subtrahend (the value to be subtracted from the minuend)

EXPLANATION:

Binary SUBTRACT interprets any leading, valid numeric characters (the digits 0 through 9, the decimal point, the Unary MINUS operator, the Unary PLUS operator, or the letter E) as the numeric value of the operand. Then, it produces a value that is the remainder after subtraction.

```
W "8 APPLES"-"4 ORANGES"
4
```

If an operand has no leading numeric characters, Binary SUBTRACT gives it a value of zero.

```
W "8 APPLES"-"FOUR ORANGES"
8
```

COMMENTS:

Binary SUBTRACT produces at least 15 significant digits of precision in its results.

RELATED:

Arithmetic Operators (Section 2.6.1)
Expression Evaluation (Section 2.7)
Numeric Expressions (Section 2.7.2)

Binary SUBTRACT (-) (Cont.)

EXAMPLES:

The following example subtracts a real numeric literal from an integer numeric literal.

```
W 122-45.6
76.4
```

The following example performs subtraction on two literals with leading digits.

```
W "12 AUTOMOBILES"-"4 TRUCKS"
8
```

INDIRECTION (@)

PURPOSE:

The indirection operator causes DSM-11 to use the value of its operand as an element in DSM-11 statements. The indirection operator may also terminate the indirection portion of an operand in partial indirection.

FORM:

@expr atom

@expr atom@(subscript)

where:

expr atom	is the expression atom whose value DSM-11 is to use
subscript	is a string that DSM-11 is to use to name an array node that descends from the variable referred by *expr atom*

EXPLANATION:

Whenever DSM-11 encounters an occurrence of indirection in a statement, it replaces the occurrence with the value and uses the value in the statement. All occurrences of indirection must evaluate to:

- One or more command arguments (argument indirection)
- A name (name indirection)
- A pattern (pattern indirection)
- A subscripted local or global variable name (partial indirection)

DSM-11 recognizes the type of indirection by its context or syntax, as follows:

1. Argument indirection

   In argument indirection, the indirection evaluates to one or more command arguments. In the following example, DSM-11 uses argument indirection to SET A(4,4,1) to the value 10.

```
S P="A(4,4,1)=10"
S @P
```

INDIRECTION (@) (Cont.)

2. Name indirection

In name indirection, the indirection evaluates to a DSM-11 name. DSM-11 defines a name as any language element that contains an alphabetic character or a percent character (%) followed by up to seven alphanumeric characters. Thus, you can use name indirection for:

- Variable names
- Line labels
- Routine names

When you use indirection to reference a named variable, the value of the indirection must be a complete variable name, including any necessary subscripts. In the following example, DSM-11 sets the variable B to the value 6.

```
S Y="B",@Y=6
```

When you use indirection to reference a line label, the value of the indirection must be a syntactically valid DSM-11 line label.

In the following example, DSM-11 sets D to the value of the line label FIG if the value of N is one, to the value of the line label GO if the value of N is two, or to the value of STOP in all other cases. Later, DSM-11 passes control to the line label whose value was given to D.

```
B   S D=$S(N=1:"FIG",N=2:"GO",1:"STOP")


    .
LV  G @D
```

When you use indirection to reference a routine name, the value of the indirection must be a syntactically valid DSM-11 routine name. In the following example, DSM-11 uses indirection to transfer control to the routine TEST2.

```
H1  S ROUT="TEST2"...


    .
    G ^@ROUT
```

INDIRECTION (@) (Cont.)

3. Pattern indirection

   In pattern indirection, the value of the indirection must be a valid pattern. See the description of the Binary PATTERN MATCH operator in this chapter for details about patterns.

4. Partial indirection

   Partial indirection requires a second indirection indicator to separate the indirected portion of the reference from a list of subscripts within parentheses. DSM-11 combines these two parts to form a single local or global reference.

   Partial indirection is syntactically different from the other three forms of indirection. Partial indirection uses two indirection operators in the form:

   @expr atom@(subscript)

   where:

   expr atom — evaluates to a local or global variable name, defined or undefined

   subscript — is a string of one or more subscripts separated by commas and enclosed in parentheses

   In the following example, DSM-11 sets the variable X(1,1) to the square of the value of X(1) if X(1) is greater than 100.

```
STRT S VAL="X(1)"
     I @VAL)100 S @VAL@(1)=@VAL*@VAL
```

COMMENTS:

In the evaluation of partial indirection, if *expr atom* refers to an unsubscripted variable, the value of the indirection is the variable name and all characters to the right of the second indirection operator (including the parentheses).

RELATED:

Indirection Operator (Section 2.6.6)
Array Structure (Section 2.4.4)

INDIRECTION (@) (Cont.)

EXAMPLES:

The following example uses argument indirection to transfer control to the routine MSB:

```
H1  S GOAR="^MSB"
    G @GOAR
```

The following example uses name indirection to transfer control to the routine MSB:

```
H2  S GOAR="MSB"
    G ^@GOAR
```

The following example sets X to the pattern for one or more digits and A to the numeric literal 4. Then, it tests whether A meets the specified pattern. Because the operand evaluates to one digit, the result is true (one).

```
S X="1N.N",A=4 W A?@X
1
```

The following example uses partial indirection to store a name and address in a local array that uses the name as the first subscript and successive integers as the second subscript. Each node of the array stores a different part of the address. The FOR statement controls the value of the integer subscript.

```
ST  S V="X(NAM)"
    S P(1)="Address: ",P(2)="City: ",P(3)="State: ",P(4)="Zip Code:"
    R "Enter Your Name: ",NAM
    F I=1:1:4 W !,P(I) R @V@(I)
```

Unary MINUS (-)

PURPOSE:

Unary MINUS gives its operand its opposite numeric interpretation.

FORM:

-operand

EXPLANATION:

Unary MINUS reverses the sign of a numerically interpreted operand.

```
W 60--60
120
```

Unary MINUS gives string-valued operands an opposite, numeric value. First, DSM-11 evaluates the string. If the string has leading numeric characters, DSM-11 uses the numerical value of these characters as the numerical value of the string.

```
W -"32 DOLLARS AND 64 CENTS"
-32
```

If the string has no leading numeric characters, DSM-11 assigns the string a numerical value of zero.

```
W -"THIRTY-TWO DOLLARS AND 64 CENTS"
0
```

Thus, you can use Unary MINUS with Binary EQUALS to test for the numeric inverse of string operands.

```
W "007"="7"
0

W -"007"=-"7"
1
```

COMMENTS:

DSM-11 gives the Unary MINUS operator precedence over the binary arithmetic operators. That is, DSM-11 first scans a numeric expression and applies any Unary MINUS operator to the operand on its right. Then, DSM-11 evaluates the expression, for example:

```
W -2CATS"_"RATS"
-2RATS
```

Unary MINUS (-) (Cont.)

If you use multiple Unary operators with an operand, DSM-11 applies them one-by-one, in right-to-left order, for example:

```
W 27---5
22
```

or:

```
W 27----5
32
```

Parentheses take precedence over Unary operators, for example:

```
W -("2CATS"_"RATS")
-2
```

RELATED:

Expression Evaluation (Section 2.7)
Unary Operators (Section 2.5)

EXAMPLES:

The following example reverses the sign of a numeric literal.

```
W -+64
-64
```

The following example gives two strings a negative numeric value and performs string arithmetic on the values.

```
W -"44 CARTONS"--"22 CARTONS"
-22
```

The following example also gives two strings a negative numeric value and performs arithmetic on the values. Because the second contains no leading digits, it evaluates to zero.

```
W -"44 CARTONS"--"TWENTY-TWO CARTONS"
-44
```

Unary NOT (')

PURPOSE:

Unary NOT inverts the truth value of the Boolean operand or operator it modifies.

FORMS:

'operand

or:

'operator

EXPLANATION:

Unary NOT with an operand inverts the truth value of the operand. If the operand was true, it now has a value of false. If the operand was false, it now has a value of true.

For example, given the statement:

```
S X=12-3,Y=0
```

then:

```
W X&Y
```

produces a false result while:

```
W X&'Y
```

produces a true result.

Unary NOT with a relational or logical operator reverses the truth value of any expression. The following is a list of the negative relational and logical operators and their meanings.

Operator	Meaning
'&	The expression is true when one or both operands are false
'!	The expression is true only when both operands are false
'=	The expression is true only when both operands are not identical strings

Unary NOT (') (Cont.)

'>	The expression is true when the left operand is less than or equal to the right operand
'<	The expression is true when the left operand is greater than or equal to the right operand
'[	The expression is true when the right operand is not contained in the left operand
']	The expression is true when the left operand does not follow the right operand in ASCII collating sequence
'?	The expression is true when the left operand is not a string in the pattern specified by the right operand

COMMENTS:

You can invert the meaning of any relational or logical operator in two ways:

1. *operand 'operator operand*

2. *'(operand operator operand)*

Both are functionally identical.

RELATED:

Expression Evaluation (Section 2.7)
Logical Operators (Section 2.6.5)
Truth-Valued Expressions (Section 2.7.3)
Unary Operators (Section 2.5)

EXAMPLES:

The following example tests the value of two local variables. If their values do not equal 12 (true), control transfers to the line labeled ADDR.

```
LET  S A=6,B=2


     G:A+B'=12 ADDR
```

The following example tests two variables with the Negative AND operator. Because one variable is true and one is false, the result is true.

```
S A=0,B=1 W A'&B
1
```

Unary NOT (') (Cont.)

The following example tests two variables with the Binary AND operator. Because the example placed Unary NOT with the false operand, the truth value was reversed and the result is true.

```
S A=1,B=0 W A&'B
1
```

Unary PLUS (+)

PURPOSE:

The Unary PLUS operator gives its operand a numeric interpretation.

FORM:

+operand

EXPLANATION:

DSM-11 interprets all operands preceded by a Unary PLUS as numeric values. If the operand has leading numeric characters, DSM-11 assigns the numeric value of the characters to the operand.

```
W +"32 DOLLARS AND 64 CENTS"
32
```

If the string has no leading numeric characters, DSM-11 assigns the string a value of zero.

```
W +"THIRTY-TWO DOLLARS AND 64 CENTS"
0
```

The Unary PLUS operator has no effect on numeric literals. Unary PLUS does not alter the sign of either positive or negative numbers.

```
S X=-23 W X
-23
```

```
S X=+-23 W +X
-23
```

COMMENTS:

You can use Unary PLUS to specify that a given subscript is a numeric subscript. For example, given the statement:

```
S A="0005"
```

Unary PLUS (+) (Cont.)

the statement:

```
S ^B(A)=X
```

defines a node with a noncanonic subscript, and the statement:

```
S ^B(+A)=X
```

defines a node with a canonic subscript.

RELATED:

Expression Evaluation (Section 2.7)
Unary Operators (Section 2.5)

EXAMPLES:

The following example evaluates a string value as a numeric value. Note that DSM-11 uses all digits as the numeric value and drops the leading zeros as it would with a numeric literal.

```
W +"0030"
30
```

The following example evaluates a string value as a numeric value. Because the string literal does not contain any leading numeric characters, its numeric value is zero.

```
W +"ABCDEFG"
0
```

Chapter 4
DSM-11 ANSI Standard Commands

This chapter describes the DSM-11 ANSI Standard commands in alphabetical order and provides examples of their use.

4.1 Introduction to DSM-11 ANSI Standard Commands

A DSM-11 command is a name for the action the command performs. DSM-11 has two types of commands:

- ANSI Standard commands
- Extended commands

ANSI Standard commands are specified in the *ANSI MUMPS Language Standard* and follow standard usage. The standard reserves the letters A through Y as the first letters of standard command names.

Extended commands, as specified in the *ANSI MUMPS Language Standard*, are implementation-specific additions to the language. Extended commands always begin with the letter Z.

For ease of use, you can abbreviate any ANSI Standard command name to its first letter.

Many DSM-11 commands can take one or more arguments. Arguments are expressions or expression atoms (for example, a function and its arguments or a variable name) that define and control the action of the command.

Some DSM-11 commands never take arguments. Their action needs no further definition than a specification of their name.

Still other DSM-11 commands take arguments only in certain circumstances. Such commands change their meaning depending on whether or not you specify an argument or argument list with them.

4.2 Command Descriptions

The following pages contain reference descriptions of all ANSI Standard DSM-11 commands. Each command description contains an explanation of the purpose, forms, and operation of the command. The descriptions are in alphabetical order for ease of referencing.

The descriptions also include one or more examples of command usage. All command-line examples are presented as they would be on entry at a terminal. All routine-line examples are presented as they would be when listed on a line printer.

BREAK

PURPOSE:

BREAK suspends execution of a routine or controls the suspension of execution from the terminal.

FORMS:

B{REAK}{:*postcond*}

B{REAK}{:*postcond*}SP*expression*

where:

postcond	is a postconditional expression
expression	is an expression that evaluates to a truth value or other system-dependent value

EXPLANATION:

The action BREAK performs depends on the form you use.

1. BREAK without arguments

   BREAK without arguments suspends the execution of a routine (allowing you to examine the effect of execution on routine variables). When you issue a ZGO command, DSM-11 resumes routine execution from the statement immediately following the BREAK. This form of BREAK is ignored unless the debugger is on (as set by ZBREAK). See the *DSM-11 User's Guide* for more discussion of the MUMPS debugger.

2. BREAK with a truth-valued expression as an argument

   BREAK with this use of the argument disables or enables recognition of CTRL/C or another specified application interrupt key. If the truth-valued expression in the argument evaluates to zero, DSM-11 disables CTRL/C recognition. If the truth-valued expression evaluates to nonzero (one), DSM-11 enables CTRL/C. See the *DSM-11 User's Guide* for more discussion of CTRL/C and the programmer abort key, CTRL/Y.

3. BREAK with a 2 or -2 as an argument

   BREAK can also have an argument of 2 or -2. In this form BREAK controls the error-processing mode. An argument of 2 enables the "Version 2" (of DSM-11) mode of error processing; an argument of -2 disables Version 2 mode error processing. See the *DSM-11 User's Guide* for more discussion of error processing.

BREAK (Cont.)

COMMENTS:

The effect of a BREAK with an argument depends on the mode in which you enter DSM-11.

When you enter DSM-11 in Programmer Mode, the system is capable of recognizing the receipt of a CTRL/C or another specified application interrupt key. This key causes an interrupt error, <INRPT>, to be generated. To disable CTRL/C recognition, you can enter a command line of:

```
B 0
```

However, each time a Programmer Mode prompt (>) is displayed CTRL/C recognition is enabled, regardless of which BREAK command was given previously.

When you enter DSM-11 in Application Mode, CTRL/C recognition is disabled. Application interrupts will remain disabled at your terminal until a routine executes the statement:

```
B 1
```

As long as the application routine does not execute such a statement, the system ignores any receipt of CTRL/C, or an alternate application interrupt key.

See the *DSM-11 User's Guide* for more information on modes and on the use of the BREAK command with arguments.

RELATED:

The ZBREAK Command
The ZGO Command
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example suspends execution of the routine INTR2 only if V is less than 20.

```
INTR2   ;ROUTINE TO CALCULATE PAYMENTS


COND    B:V<20
```

The following example unconditionally suspends the execution of the routine at the line labeled TAG3. The last statement on the line (W !,A(I)) is not

BREAK (Cont.)

executed because it is after the BREAK. It is executed as soon as you issue a ZGO.

```
TAG3 S A(I)=L B  W !,A(I)
```

(Note the two spaces between the unconditional BREAK and the WRITE statement.)

The following example disables recognition of CTRL/C or another application interrupt key.

```
B 4
```

The following example enables recognition of CTRL/C.

```
B 1
```

The following example enables Version 2 of DSM-11 compatible error processing.

```
S 2
```

The following example disables Version 2 compatible error processing.

```
B
```

CLOSE

PURPOSE:

CLOSE releases ownership of one or more specified input/output devices (and, optionally, performs device-dependent functions prior to that release).

FORM:

C{LOSE}{:*postcond*}(SP)*argument*,...

In which *argument* can be one of the following:

device{:*params*}

@expr atom

where:

postcond	is a postconditional expression
device	is a device specifier
params	are one or more expressions that indicate any action to take upon release of the device
@expr atom	is an indirect reference that evaluates to one or more CLOSE arguments

EXPLANATION:

CLOSE deallocates the device(s) specified in the argument(s) and makes them available to another user. Each argument for the CLOSE command consists of a valid DSM-11 device specifier and, optionally, its modifying parameters.

The device specifier you use can be any valid device. You can make the specification by indirection.

The parameters you use with CLOSE consist of one or more expressions that specify any special action to take on release of the device. The values you can use for parameters depend on the device you are releasing. If you specify more than one parameter, you must enclose the parameters in parentheses and separate the parameters with colons.

See the discussion of I/O devices in the *DSM-11 User's Guide* for more information on device specifier and device-specific parameters.

COMMENTS:

CLOSE (Cont.)

You should keep the following points in mind when you use the CLOSE command:

1. DSM-11 always resets your principal device as your current device when you close a device you have used for I/O. Thus, after you execute a CLOSE, the $IO special variable contains the device specifier for your principal device.

   The $IO special variable contains the specifier of the device you are currently using for input and output. Thus, $IO has the value of either your principal device or your current device.

   Your principal device is the device the system opens when you log in and uses for I/O by default. You can refer to your principal device as device 0. The current device is any device you open with an OPEN command and to which you direct I/O operations with a USE command. The system ignores a CLOSE 0 command. If you CLOSE $IO and $IO is set to your principal device, the effect is the equivalent of a HALT command.

2. DSM-11 does not return control to your routine until the device is closed. If the device you are closing is an output device, DSM-11 does not close the device until all current output operations are complete.

3. When you execute a HALT command, DSM-11 closes (releases your ownership of) all still opened devices.

RELATED:

The OPEN Command
The USE Command
The ZUSE Command
The $IO Special Variable
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following example reads the data from the first level of the global ^NAM and writes it on the specified device.

```
ST   R !,"ENTER DEVICE SPEC",DEV
     O DEV S A="" U DEV
A1   S A=$O(^NAM(A)) I A="" C DEV Q
     W A," " W:$D(^NAM(A))#10 ^NAM(A) W ! G A1
```

DO

PURPOSE:

DO directs control to a specified line in the current routine or to any specified line and/or routine. Control returns to the point immediately following the DO argument.

FORM:

D{O}{:*postcond*}

D{O}{:*postcond*}␣*argument*,...}

In which *argument* can be one of the following:

entry ref{:*postcond*}

@expr atom

where:

postcond	is an postconditional expression
entry ref	entry is an entry reference specifying the line and/or routine
@expr atom	is an indirect reference that evaluates to one or more DO arguments

EXPLANATION:

DO is a control command that gives you a generalized capability of executing DSM-11 routines. It allows you to execute the routine currently in memory or any other routine.

To execute the current routine in memory, use as an argument an entry reference to the first line you want DSM-11 to execute. DSM-11 begins executing the routine at the line you specify.

To execute a routine in your routine directory use as an argument an entry reference that includes the routine name preceded by a circumflex. DSM-11 loads the routine and begins execution at the first executable line or (if you included a line label on line label and offset) at the line you specify.

When DSM-11 reaches a QUIT statement or the end of the routine, it returns control to the point immediately following the DO argument.

DO without an argument is used in block-structured programming. The DO introduces an indented execution block that begins on the following line.

DO (Cont.)

When the end of the execution block is reached, or a QUIT statement is reached within the execution block, control returns to the next command or line following the DO at the same level as the DO. An execution block ends either at the end of the routine or at the next lower level routine line.

The value of the $TEST special variable is never altered by the execution of an argumentless DO. This is different from DO with an argument, which does not preserve $TEST.

See Section 1.7 for more discussion of block-structured programming.

COMMENTS:

Keep the following points in mind when using the DO command:

1. You can use indirection for any form of DO command arguments.

   For a DO argument:

```
A   S X="^A,^B"
    D @X
```

   For a line label (ADDR):

```
B2   S A="ADDR"
     D @A
```

   For a line label and offset (ADDR + 1):

```
S1   S A="ADDR"
     D @A+1^ROUT
```

   For a routine name (TEST):

```
B1   S:ANS="TEST" R="TEST"
     D ^@R
```

   For both line labels and routine names (START, TEST):

```
NEW   S X="START",Y="TEST"
      D @X^@Y
```

2. You can use postconditional expressions with both DO and its arguments. If a command postconditional is false, DSM-11 ignores the DO statement. If an argument postconditional is false, DSM-11 ignores the argument to which it is appended.

DO (Cont.)

3. When you are using DO without an argument, the structure of the routine must follow that indicated in Section 1.7. Each line of a subroutine introduced by a DO must be indented with one or more periods (after the tab stop) than the previous line. Additional subroutines can be introduced by argumentless DOs; each higher level subroutine must be indented an additional period after the tab stop at the beginning of each line. A GOTO must only be used to transfer control to the same level and within the same block. In the following example, the transfers to LABEL1 are all within the second level.

```
TEST   S X=5 DO
LABEL1 .S Y=7 R "INPUT: ",BVAL


       .I NVAL=0 GOTO LABEL1
       .DO
LABEL2 ..S J=1 I M=0


       ..GOTO LABEL2
       .I NVAL=0 GOTO LABEL1


       W !,"Result is: ",LVAL
```

RELATED:

The GOTO Command
The HALT Command
The QUIT Command
The XECUTE Command
Entry References (Section 1.6.2)
Extensions to Routine Structure (Section 1.7)
The Indirection Operator (Section 2.6.6)
Line Labels (Section 1.4.2.1)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example transfers control to a line labeled FIX if the variable M is greater than 20. Note that periods in this case indicate omitted lines of the routine, not block structuring.

DO (Cont.)

```
ADD   ;ADD ROUTINE


      .
      D:M>20 FIX
```

The following example loads and executes the routine TABUL from the routine directory. Because the DO command was entered as a command line, DSM-11 returns its prompt after execution of the routine. Note that periods in this case indicate omitted lines of the routine, not block structuring.

```
D ^TABUL


.
>
```

The following example loads TABUL from the routine directory and begins execution on the line labeled B2.

```
D B2^TABUL
```

The following example illustrates multiple DO arguments. The routine line specifies that, if the first character in the string represented by ANS is Y, DSM-11 is to execute the routines AD1 and AD2. If, in addition, X is less than four, DSM-11 is to execute AD3.

```
BR1   D:$E(ANS,1)="Y" ^AD1,^AD2,^AD3:X<4
```

The following example shows a three-level block-structured routine:

```
ABC   W !,"Begin level 1" F I=1:1:3 D
      .W !,"Begin level 2" F J=1:1:3 D
      ..W !,"Level 3 line #",I,".",J
      W !,"End of level 1" Q
```

ELSE

PURPOSE:

ELSE conditionally executes the statements following it on the same line depending on the truth value in the $TEST special variable.

FORM:

E{LSE}

EXPLANATION:

When DSM-11 encounters an ELSE command, it tests the value of the $TEST special variable. If the value of $TEST is one, DSM-11 does not execute the remainder of the line to the right of ELSE. If the value of $TEST is zero, DSM-11 does execute the remainder of the line. $TEST contains the Boolean result of the last IF command, timed READ, timed LOCK, timed OPEN, or JOB command. For instance if a READ cannot be completed within the time indicated in a conditional timeout, then $TEST is 0. If the argument evaluates to 0 for an IF command then the commands following the IF are not executed; and $TEST is set to 0. An ELSE on the following line would then be executed (because $TEST is 0).

ELSE does not affect the value of $TEST.

COMMENTS:

Keep two points in mind when you use ELSE:

1. DSM-11 can never execute an ELSE on the same line as an IF statement unless an intervening statement that does not contain an IF command resets the value of $TEST. For example, in the routine fragment:

```
TES   I AGE(I)>19 W !,"GOOD" E   W "BAD"
      E   W !,"NO GOOD" Q
      W !,"DONE"
```

   the ELSE statement on the second line is the only one that can execute.

   When the IF statement is true, DSM-11 sets $TEST to one, executes the first WRITE statement, and ignores the WRITE statement following the ELSE. When the IF statement is false, DSM-11 ignores the statements that follow it and executes the second line.

   If there is an intervening statement that resets $TEST, DSM-11 can execute an ELSE on the same line as an IF. Consider the following example:

ELSE (Cont.)

```
N  I A R X:10 E  G RETRY
```

If A is true, DSM-11 sets $TEST to true (one) and executes the READ. If the timeout occurs, DSM-11 executes the ELSE and the subsequent GOTO statement.

2. In many situations, you can substitute postconditional expressions and the $SELECT function for IF and ELSE statements. For example, the routine lines:

```
B1   I A)14 W "YES"
     E  W "NO"
```

and:

```
B1   W:A)14 "YES"
     W:A')14  "NO"
```

and:

```
B1   W $S(A)14:"YES",1:"NO")
```

produce the same output.

In this particular case, using the $SELECT function would be the best choice. It executes faster because DSM-11 must make one relational test instead of two.

(Neither postconditional expressions nor $SELECT affect $TEST.)

RELATED:

The IF Command
The $SELECT Function
The $TEST Special Variable
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following example attempts to open a specified device. If the device is busy (owned by another user), DSM-11 sets $TEST to 0 and executes the statement to the right of the ELSE.

```
DEV  R !,"USE DEVICE ?",DEV
     O DEV::0 E  W !,"DEVICE BUSY"
```

FOR

PURPOSE:

FOR controls the repeated execution of all remaining statements following it on a line for successive values of a local variable. (The remaining statements are called the scope of the FOR statement.)

FORMS:

F{OR}␣*local*=*parameter*,...

In which *parameter* can be:

expression

start:step

start:step:stop

where:

local	is a local variable name or an indirect reference that evaluates to a local variable name
expression	is an expression (not necessarily numeric)
start	is a numeric-valued expression specifying the initial value given to the local variable
step	is a numeric-valued expression specifying the value added to that of the local variable on each execution of the remaining statements on the line
stop	CFI is a numeric-valued expression specifying the upper limit (largest if *step* is positive, smallest if *step* is negative) of value to be added to the local variable

EXPLANATION:

The meaning of FOR depends on the form of the FOR parameter you use.

1. *local*=*expression*

   DSM-11 gives *local* the value of *expression*. Then, if DSM-11 does not encounter any control statements (such as QUIT or HALT), it executes all other statements on the same line and evaluates the next line.

2. *local*=*start:step*

DSM-11 performs the following actions:

a. It gives *local* the value of *start*.

b. It executes the remaining statements on the line.

c. It adds the value of *step* to the value of *local*.

d. It returns to step b.

This form of the FOR parameter creates an endless loop. To stop execution, use a GOTO or QUIT command in the scope of the FOR statement (or CTRL/C if it is enabled) to transfer control elsewhere.

In the following example, DSM-11 gives I the values 1 to 100 and quits.

```
A1 F I=1:1 W I Q:I>100
```

3. *local*=*start*:*step*:*stop*

DSM-11 performs the following action if *step* is positive or zero:

a. It gives *local* the value of *start*.

b. It determines if the value of *local* is now greater than that of *stop*. If so, DSM-11 goes to the next FOR parameter or to the next line if no other FOR parameters are present. If not, DSM-11 goes to step c.

c. It executes the remaining statements on the line.

d. It determines if the value of *local* incremented by the value of *step* is greater than the value of *stop*. If so, DSM-11 goes to the next FOR parameter or to the next line if no other FOR parameters are present. If not, DSM-11 goes to step e.

e. It adds the value of *step* to the current value of *local*.

f. It returns to step c.

DSM-11 performs the following action if *step* is negative:

a. It gives *local* the value of *start*.

b. It determines if the value of *local* is less than *stop*. If so, DSM-11 goes to the next FOR parameter or to the next line if no other FOR parameters are present. If not, DSM-11 goes to step c.

c. It executes the remaining statements on the line.

d. It determines if the value of *local* incremented by the value of *step* is less than the value of *stop*. If so, DSM-11 goes to the next FOR parameter or to the next line if no other FOR parameters are present. If not, DSM-11 goes to step e.

e. It adds the value of *step* to the current value of *local*.

f. It returns to step c.

This form of the FOR parameter terminates itself. When an increment to the value of *local* would give it a value greater than *stop* when *step* is positive or a value less than *stop* when *step* is negative, DSM-11 completes processing the FOR parameter.

During the execution of the FOR command, *start*, *step*, and *stop* are evaluated first. For example, the following statements give the result of 123 (as shown) rather than simply a result of 1:

```
SET I=3
FOR I=1:1:I WRITE I
123
```

COMMENTS:

Keep in mind the following points when you use the FOR command:

1. The numeric expressions you use in a FOR parameter can be either integer or decimal values.

2. You cannot use postconditional expressions with either the FOR command or its parameters.

3. You can include more than one parameter in each FOR statement. For example, the statement:

   ```
   F I=1:1:3,7,20:10:100
   ```

   has three parameters (1:1:3, 7, and 20:10:100).

   When DSM-11 executes this statement:

   a. It sets I equal to 1, 2, and 3.

   b. It executes the second parameter and sets I to the value of 7.

   c. It executes the third parameter, sets I to the value of 20, and increments the value of I by 10 until I reaches 100.

FOR (Cont.)

4. You cannot include more than one complete argument with a FOR command. That is, a FOR statement such as:

```
F I=1:1:300,J=1:20:1000...
```

   is invalid.

5. When you use the *start:step* form of the FOR parameter, DSM-11 can never execute any additional parameters to the right. For example, in the statement:

```
F I=1,2:2,100:1:200...
```

   the parameter 100:1:200 is never executed because the QUIT needed to terminate the parameter 2:2 (somewhere at a later point on the line) also terminates the FOR.

6. If you use more than one FOR statement on a line, the rightmost FOR is considered to be nested in the FOR to its left. One execution of the scope of the left (outer) FOR includes one complete pass through the inner FOR's parameter list.

   If the scope of the inner FOR includes a QUIT statement, DSM-11 immediately terminates the inner FOR. DSM-11, however, only terminates that particular iteration of the outer FOR.

   If the scope of the inner FOR includes a GOTO statement, DSM-11 terminates all FOR statements to the left of the GOTO. It then transfers control to the specified point.

7. You cannot use argument indirection to an entire FOR argument. Such an attempt results in an error. You can, however, use name indirection to each element in the FOR parameter.

```
S A="1",B="3",C="A",D="B"
FI=@C:@C:@D W ?$X+2,I
 1  2  3
```

RELATED:

The GOTO Command
The QUIT Command

EXAMPLES:

The following example uses nested FOR statements to display a local array.

```
F I=1:1:10 F J=1:1:5 W A(I,J)
```

FOR (Cont.)

The following example uses negative integers in the FOR parameter.

```
F X=4:-3:-3 W X
41-2
```

The following example shows that you can change the value of the local variable you use as an index within the scope of the FOR.

```
F J=1:2 10 S J=J-1 W J
0123456789
```

The following example displays the values of the special variables. It also demonstrates that the *local=expression* form of the FOR argument is not evaluated numerically.

```
WR   F I="I","S","X","Y" W:$X>10 ! W "   $",I,"=".@("$"_I)
```

The following example prompts for interactive executions of the routine ROUT. If X is zero, DSM-11 never executes ROUT.

```
TEST1   R !,"HOW MANY ITERATIONS?",X F I=1:1:X D ^ROUT
```

GOTO

PURPOSE:

GOTO transfers control to a specified line or routine. Control does not return to the point immediately following the GOTO argument.

FORM:

G{OTO}{:*postcond*}␣*argument*,...

In which *argument* can have the following forms:

entry ref{:*postcond*}

expr atom

where:

postcond	is a postconditional expression
entry ref	is an entry reference specifying the line and/or routine to which control is to be transferred
@expr atom	is an indirect reference that evaluates to one or more GOTO arguments

EXPLANATION:

GOTO is a control command that gives you a generalized transferral ability. You can use it to transfer control to another section of the routine currently in memory or to any other routine.

To transfer control to a line in the current routine in memory, use an argument consisting of an entry reference to the line you want DSM-11 to execute. DSM-11 transfers control to the line you specify. The entry reference can specify the same line that contains the GOTO.

To transfer control to a routine not currently in memory, use an argument consisting of an entry reference that includes the name of that routine preceded by a circumflex. DSM-11 loads the specified routine and begins execution at the first executable line or (if you included a line label or a line label and offset in the entry reference) at the line you specify.

After transfer when DSM-11 encounters a QUIT command or the end of the routine, DSM-11 exits from the routine. It does not return control to the argument or statement immediately to the right of the GOTO argument that caused the transfer. If you want to return control to the original point, use the DO command.

GOTO (Cont.)

COMMENTS:

Keep the following points in mind when you use the GOTO command.

1. You can use postconditional expressions with both GOTO and its arguments. If a GOTO postconditional is false, DSM-11 ignores the entire statement. If an argument postconditional is false, DSM-11 ignores the argument.

2. You can use multiple arguments with the GOTO command. However, because DSM-11 does not return control to the point immediately following the argument that caused the transfer, only the first argument that has no postconditional or that has a true postconditional is ever executed.

3. You can use indirection for any form of GOTO command arguments.

   For a GOTO argument:

   ```
   A1   S X="^A:'ANS,^B"
        G @X
   ```

   For a line label (ADDR):

   ```
   B2   S A="ADDR"
        G @A
   ```

   For a line label and offset (ADDR + 1):

   ```
   S1   S A="ADDR"
        G @A+1^ROUT
   ```

   For a routine name (TEST):

   ```
   B1   S:ANS="TEST" R="TEST"
        G ^@R
   ```

   For both line labels and routine names (START, TEST):

   ```
   NEW  S X="START",Y="TEST"
        G @X^@Y
   ```

4. If you use a GOTO in a block-structured routine, you can only transfer control within the same execution block. See the DO command and Section 1.7 for more discussion of block-structured programming.

RELATED:

The DO Command
The FOR Command

GOTO (Cont.)

The QUIT Command
The XECUTE Command
Entry References (Section 1.6.2)
Extensions to Routine Structure (1.7)
The Indirection Operator (Section 2.6.6)
Line Labels (Section 1.4.2.1)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example transfers control to the fourth line following A1 if L is less than 10.

```
B6      G:L<10 A1+4
```

The following example transfers control to the first executable line in the routine ROUT.

```
J1      G ^ROUT
```

The following example transfers control to line A if X is less than 10, to line D if X is greater than 10, and to line H if X equals 10.

```
TX      G A:X<10,D:X>10,H
```

HALT

PURPOSE:

HALT ends your use of DSM-11.

FORM:

H{ALT}{:*postcond*}

where:

postcond is a postconditional expression

EXPLANATION:

When you enter an unconditional HALT, DSM-11 terminates your job. It also unlocks all local and global nodes you locked and closes all devices you own.

If you use a postconditional expression with HALT, DSM-11 executes the HALT only if the postconditional is true.

RELATED:

The CLOSE Command
The LOCK Command
The HANG Command
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example executes an unconditional HALT.

```
H
```

The following example executes a conditional HALT. DSM-11 executes the HALT only if A is greater than B. In any other case, it executes the WRITE statement.

```
STOP  H:A>B  W "NO HALT"
```

HANG

PURPOSE:

HANG suspends execution for a specified number of seconds.

FORM:

H{ANG}{:*postcond*}␣*argument*,...

In which *argument* can have the following forms:

seconds

@expr atom

where:

postcond	is a postconditional expression
seconds	is a real- or integer-valued expression specifying the number of seconds to suspend execution
@expr atom	is an indirect reference that evaluates to one or more HANG arguments

EXPLANATION:

HANG suspends execution of a routine for the number of seconds you specify in the argument. DSM-11 resumes routine execution at the statement following the HANG.

If the integer expression you specify is zero or negative, DSM-11 ignores the HANG. If you use a decimal numeric expression, DSM-11 truncates it to an integer.

COMMENTS:

Keep the following points in mind when you use the HANG command:

1. You cannot use a postconditional expression with a HANG argument.

2. You can use indirection with HANG. The form of indirection is:

   @expr atom

   The replacement after applying indirection must evaluate to one or more HANG arguments. Thus the statements:

```
S B="2",A="B"
H @A
```

or:

```
S A="2"
H @A
```

are valid.

RELATED:

The HALT Command
The QUIT Command
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example unconditionally suspends the routine for two seconds.

```
B3    H 2
```

The following example prompts for input and waits for an answer. If the answer is not entered, DSM-11 sounds the bell on the terminal, hangs for one second, and again waits for the answer.

```
      W "TELL ME WHAT TO DO?",!
LOOP  R ANS:10 E  W *7 H 1 G LOOP
```

The following example issues a prompt for you to perform multiplication. If the sum of the values to be multiplied is greater than 100, the example hangs for ten seconds before prompting for an answer.

```
TEST  W "MULTIPLY",A,"TIMES",B H:A+B>100 10
      R !,"WHAT IS YOUR ANSWER?",ANS
```

IF

PURPOSE:

IF permits the conditional execution of the statement or statements that follow it.

FORMS:

I{F}

I{F}␣*argument*,...

In which *argument* can be one of the following:

truth value

@expr atom

where:

truth value	is a truth-valued expression
@expr atom	is an indirect reference that evaluates to one or more IF arguments

EXPLANATION:

IF makes the execution of all statements that follow it dependent on the value of the $TEST special variable.

1. IF without arguments

   IF without arguments is the inverse of ELSE. Execution of the statements following it depends on the existing value of $TEST set by a previous statement.

   If $TEST has the value of true (one), DSM-11 executes all statements to the right of the IF. If $TEST has the value of false (zero), DSM-11 ignores all statements to the right of the IF and executes the next line.

2. IF with arguments

   Execution of the statements following an IF with arguments depends on the value of $TEST set as a result of those arguments. When you enter an IF command with one argument, DSM-11 evaluates the argument as a Boolean expression and places the result in $TEST.

IF (Cont.)

If the value of the argument is true (one), DSM-11 executes all statements to the right of the IF. If the value of the argument is false (zero), DSM-11 ignores all statements to the right of the IF and executes the next line.

When you enter an IF command with multiple arguments, DSM-11 evaluates each argument left to right and performs a logical AND operation on each value. If all the arguments are true, DSM-11 executes the statements to the right of the IF. If any argument is false, DSM-11 ignores the statements to the right of the IF and executes the next line.

COMMENTS:

Keep the following points in mind when you use the IF command:

1. You cannot use postconditional expressions with either the IF command or with any of its arguments.

2. You can use argument indirection with the IF command. For example, if a routine contained the statements:

```
S Y="3"

S X="1,2,@Y"
```

   then the following statements write the string "TEST":

```
I @X W "TEST"
```

3. Using commas to perform a logical AND operation on alternatives can reduce the amount of computation since only the first alternative may be evaluated. For example, of the following two statements, the first statement is more efficient than the second:

```
I A=4,B=3...

I A=4&(B=3)...
```

   (Note that the second example requires the parentheses.)

RELATED:

The ELSE Command
The $TEST Special Variable

EXAMPLES:

The following example requests terminal input. If the operator types a carriage return (a null string), the routine stops running.

IF (Cont.)

```
QUEST   W !,"CONTINUE?" R !,ANS
        I ANS="" W !,"EXIT" Q
```

The following example executes the routine ROUT only if A equals 4 and B equals 6.

```
PL      I A=4,B=6 D ^ROUT...
```

The following example illustrates the use of an IF without arguments with a timed READ.

```
IN      R !,"ANSWER WITHIN 30 SECONDS",ANS:30
        I W "ANSWER RECEIVED" G A1
```

The following example shows both the use of an IF without arguments and the use of the ELSE command. If A is greater than five, DSM-11 prints BIGGER THAN FIVE; otherwise, it prints SMALLER.

```
TEST    I A>5 W !,"BIGGER"
        E   W !,"SMALLER"
        I   W !,"THAN FIVE"
```

JOB

PURPOSE:

JOB starts a specified routine in a new partition.

FORM:

J{OB}{:*postcond*}␣{*argument*}

In which argument can be one of the following:

entry ref{ *UCI*{,*volume set*}]}{:*size*}{:*timeout*}

@expr atom

where:

postcond	is a postconditional expression
entry ref	is an entry reference specifying the line and/or routine to execute
UCI	is a three-character string that designates a UCI
volume set	is a three-character string that designates a volume set
size	is a numeric expression that specifies the size of the partition in 512-byte increments
timeout	is a timeout
@expr atom	is an indirect reference that evaluates to one or more JOB arguments

EXPLANATION:

JOB can start another routine executing in another partition. In DSM-11, such a detached routine is called a background job.

If you use only a routine name as an entry reference, DSM-11 begins the routine from the first executable statement. If you also specify a line label or line label and offset in the entry reference, DSM-11 begins execution at the line indicated.

Normally, you can start only routines that are in your UCI (User Class Identifier). However, if you are running under the System Manager's account, you can start a routine filed under any UCI and volume set by specifying the

JOB (Cont.)

UCI and volume set. The UCI should be a string value that specifies the UCI as it appears in the system UCI Table.

See the *DSM User's Guide* for more information on the UCI and the UCI Table.

If you need a particular size partition for the routine, indicate this with the optional *size* argument. Give *size* an integer value that, multiplied by 512, indicates the size of the needed partition in bytes. The DSM-11 system, however, expects the partition size to be in increments of 1K bytes (1026 bytes). If you specify an odd number of 512-byte increments, such as 5, in the JOB command, the system rounds up the number of increments to an even number, resulting in a partition size that fits the 1K-byte expectation. For example, the 5 (JOB command) increments would be rounded up to 6 increments, giving a partition size of 3K bytes.

If you do not indicate a size for the partition, DSM-11 attempts to open a default-sized partition. (See the *DSM User's Guide* for a specification of the default size.)

If DSM-11 is not able to open a partition of the specified size (or of the default size if you did not specify a size), it attempts to open a larger partition. If DSM-11 is able to open a partition and start the routine, it sets $TEST to true (one) and the $ZB special variable to the job number assigned to the started routine. If DSM-11 is not able to open a partition and start the routine, it sets $TEST to false (zero) and $ZB to zero.

COMMENTS:

DSM-11 imposes certain conditions on the partition opened by JOB:

1. The symbol table for the partition is empty. No variables defined in the initiating job are carried over into the newly opened partition.

2. The partition does not own any devices. The partition has the same principal device as the partition that initiated it. If the partition does not OPEN and USE a device, output is defaulted to the principal device.

Thus, when the routine running in the new partition commits an error and the value of $ZTRAP is null, DSM-11 displays the error message on your principal I/O device. The routine terminates without giving you a chance to examine the partition.

RELATED:

The OPEN Command
The USE Command
The $TEST Special Variable

JOB (Cont.)

Entry References (Section 1.6.2)
Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example attempts to open a default-sized partition to run the routine ^ROUT, and it displays the contents of $TEST to determine if the attempt was successful.

```
J ^ROUT
W $T
1
```

The following example attempts to start the routine ADD in an 8192-byte partition. If DSM-11 is not able to start ADD because no 8192-byte or larger partition is available, DSM-11 sets $TEST to zero and displays the message ADD NOT STARTED.

```
J ^ADD:16 E  W !,"ADD NOT STARTED"
```

The following example attempts to start a routine called TEST at the line labeled START in a 4096-byte partition. The string in square brackets is the three-letter code for the UCI under which the routine is stored.

You can use this syntactical form only from the System Manager's account.

```
J START^TEST["ASG"]:8
```

KILL

PURPOSE:

KILL deletes the specified local or global variables.

FORMS:

K{ILL}{:*postcond*}

K{ILL}{:*postcond*}(SP)*argument*,...

In which *argument* can be one of the following:

name

(*local*{,...})

@expr atom

where:

postcond	is a postconditional expression
name	is the name of a local or global variable with or without subscripting
local	is the name of a local variable or an indirect reference that evaluates to a variable name or to another indirect reference
@expr atom	is an indirect reference that evaluates to one or more KILL arguments

EXPLANATION:

KILL without arguments removes all local variables from your partition. It has no effect on global variables.

When you kill a variable, the value of that variable is undefined. An application of the $DATA function to that variable produces a value of zero.

The effect of KILL with arguments depends on the argument form you use.

1. *name*

   This form of the KILL argument is a selective kill. It deletes only the local or global variable you specify.

If the variable you specify does not exist, the kill has no effect. If the variable you specify is unsubscripted, DSM-11 also deletes all nodes (elements) of any subscripted variable with the same name.

If the variable you specify is subscripted, DSM-11 also kills its descendant nodes. For example, if an array contains the nodes:

N(1,2,3)
N(1,2,3,4)
N(1,2,3,7)
N(1,3,1)

and you kill N(1,2,3), DSM-11 also kills N(1,2,3,4) and N(1,2,3,7).

After you kill a node that has no siblings, DSM-11 changes the descendant attribute of the parent node. If the parent is a logical pointer node that contains no data, DSM-11 deletes the parent.

For example, when you kill all descendants of ^N(1,2), DSM-11 clears ^N(1,2) of its attribute of having descendants. Thus, when you apply $DATA to ^N(1,2), the result is one.

If ^N(1,2) is a logical pointer node, ^N(1,2) disappears. If you apply $DATA to ^N(1,2), the result is zero.

See the description of $DATA in Chapter 6 for more information.

2. (*local*{,...})

The second form of the KILL argument is an exclusive kill. It deletes all local variables but those named in the argument and any descendants those variables might have. The local variables you specify should be unsubscripted.

This form of KILL does not affect global variables.

COMMENTS:

If you use a postconditional expression with the KILL command, DSM-11 executes the KILL only if the postconditional expression is true.

RELATED:

The SET Command
The $DATA Function
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

KILL (Cont.)

The following example deletes the local variable A and any descendants it may have.

```
K A
```

The following example deletes the global node ^A(3) and any descendants it may have (for example ^A(3,1), ^A(3,1,3), ^A(3,4,5,6)).

```
K ^A(3)
```

The following example deletes all local variables.

```
K
```

The following example deletes all local variables except A, C, and ANS and any descendants they may have.

```
K (A,C,ANS)
```

The following example uses a postconditional expression. If A equals 1, DSM-11 kills ANS(A) and any descendants it may have.

```
K:A=1 ANS(A)
```

The following example uses argument indirection to kill the local variables Y and Z.

```
S X="2",Y="(X)",Z="2"
K @Y
```

LOCK

PURPOSE:

LOCK makes a specified variable or specified nodes of a variable unavailable for locking by another user.

FORMS:

L{OCK}{:*postcond*}

L{OCK}{:*postcond)*}␣*argument*{:*timeout*},...

In which *argument* can be one of the following:

name

(*name*{,...})

@expr atom

where:

postcond	is a postconditional expression
name	is the name of a local or global variable with or without subscripting
@expr atom	is an indirect reference to one or more arguments
timeout	is a timeout that specifies how many seconds DSM-11 is to try to lock the variable

EXPLANATION:

The LOCK command locks and unlocks both global and local variables. When you lock a variable, DSM enters its name into a table that prevents any other user from entering a LOCK command for that variable.

If you lock a local variable, DSM-11 locks all local variables with that name throughout the system. For example, the statement:

```
L A
```

executes a system-wide lock of all local variables named A.

If you lock a global variable, DSM-11 prevents any other user in your UCI from locking that variable. For example, the statement:

LOCK (Cont.)

```
L ^A
```

performs a UCI-wide lock of the global ^A.

When you unlock the variable, DSM-11 removes the variable name from the table. The variable is free for locking by other users.

Used as a convention, LOCK can be a flag to other users that indicates that you are modifying a variable or adding nodes to an array. If all users follow the practice of locking a variable before working on it, you can avoid two or more users making conflicting modifications to the same array.

LOCK, without arguments, unlocks all previously locked variables. It does not lock any new variables.

The effect of LOCK with arguments depends on the argument form you use.

1. *name*

   LOCK prevents another user from locking a variable or specified portion of an array until you unlock it. DSM-11 unlocks that local or global variable when any of the following actions occur:

   - You execute an argumentless LOCK.
   - You issue a LOCK to lock a different portion of the variable array or a different variable.
   - You execute a HALT.

2. (*name*{,...})

   LOCK locks all the variables you specify or waits until all the variables you specify are free for locking. That is, if you issue the LOCK statement (as in the following statement) when the variable ^E is already locked, LOCK waits until E is available before locking the variables.

   ```
   L (^A,^D,^E)
   ```

   If you do not enclose multiple LOCK arguments in parentheses, DSM-11 interprets the statement as multiple LOCK directives rather than as one LOCK directive with multiple arguments.

   For example, the following statement:

   ```
   L (^A(1,2,3),^B(4),^C(3,1))
   ```

   is interpreted as locking ^A(1,2,3), ^B(4), and ^C(3,1).

   On the other hand, the following statement:

```
L ^A(1,2,3),^B(4),^C(3,1)
```

is interpreted as:

a. Locking ^A(1,2,3).

b. Unlocking ^A(1,2,3) and locking ^B(4).

c. Unlocking ^B(4) and locking ^C(3,1).

Thus, the statement only locks ^C(3,1).

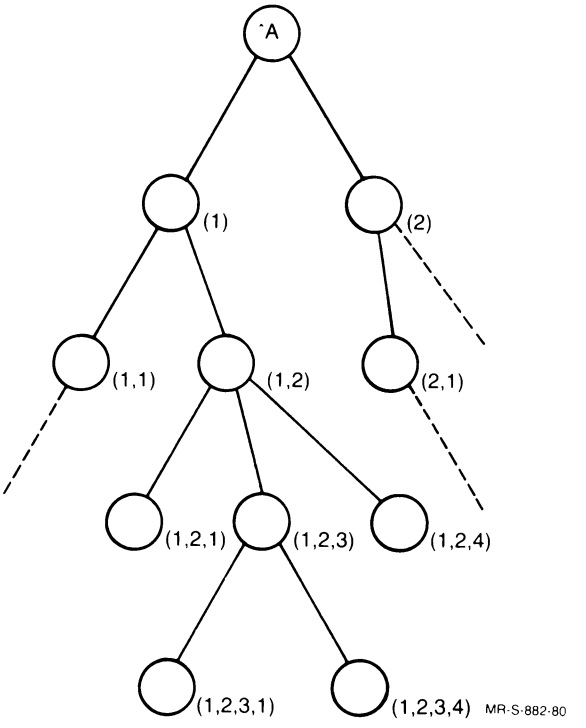
You must always use full variable references, or indirection to full references, for LOCK arguments. DSM-11 does not allow naked references with LOCK.

If you use an unsubscripted name as the argument for either form of LOCK, DSM-11 prevents any other user from locking any node of that variable until you unlock it.

If you use a subscripted node as the argument for either form of LOCK, DSM-11 locks the node and all descendants of that node. DSM-11 also makes any ancestors of that node unavailable for locking.

Consider the global ^A in Figure 4-1.

Figure 4-1: Locked Global



LOCK (Cont.)

If you lock the node ^A(1,2,3), DSM-11 also locks the following nodes:

^A(1,2,3,4)
^A(1,2,3,1)

These nodes are the descendants of ^A(1,2,3)

DSM-11 also makes the following nodes unavailable for locking:

^A(1,2)
^A(1)
^A

These nodes are the ancestors of ^A(1,2,3).

The following nodes, however, are available for locking:

^A(2)
^A(1,1)
^A(2,1)
^A(1,2,1)
^A(1,2,4)

COMMENTS:

Keep the following points in mind when you use the LOCK command:

1. LOCK is a convention. It does not prevent multiple, simultaneous accesses of a variable. Any user who does not follow the convention of issuing LOCK commands and avoiding work on variables locked by others can modify a variable for which you have issued a LOCK.

2. To avoid a potentially long suspension of execution if the variable you are attempting to lock is already locked by another user, use a timeout. If DSM-11 cannot lock the variable before the timeout, it sets $TEST to false (zero) and resumes execution. If DSM-11 can lock the variable before the timeout, it locks the variable, sets $TEST to true (one), and resumes execution.

   DSM-11 resumes execution as soon as it can lock the variable. It ignores any time remaining in the timeout.

3. Even if a timed lock with arguments reaches a timeout before locking the specified variable(s), it does unlock any previously locked variables. For example, if you successfully lock ^A and ^B:

```
L (^A,^B):10
```

and then later try a timed lock on ^C that cannot lock ^C before the timeout:

```
L ^C:10
```

the second LOCK still unlocks ^A and ^B.

RELATED:

The ZALLOCATE Command
The ZDEALLOCATE Command
The $TEST Special Variable
Global Variables (Section 2.4.2)
The Indirection Operator (Section 2.6.6)
Local Variables (Section 2.4.1)
Postconditional Expressions (Section 2.7.4)
Timeout Expressions (Section 2.7.5)

EXAMPLES:

The following example locks the global ^LIS. No other user can now lock the unsubscripted global name ^LIS or any subscripted global node that is part of ^LIS. Any globals previously locked by this job are unlocked.

```
LCK   L ^LIS
```

The following example locks the nodes ^AS(1,3) and ^J(4,1) and all their descendants. It also makes their ancestors in a direct line to the roots ^AS and ^J unavailable for locking.

```
LCK2  L (^AS(1,3),^J(4,1))
```

The following example unlocks all locked variables.

```
L
```

The following example tries for four seconds to lock global ^A. If the attempt is not successful, control passes to the line labeled B2. If the attempt is successful, control passes to the next sequential line.

```
TES3  L A:4 E  G B2
```

The following example locks the nodes ^A(1,1) and ^A(1,2,3,4).

```
RT    L ^A(1,2)
      L (^A(2,1),^A(1,1),^A(1,2,4))
      L (^A(1,1),^A(1,2,3,4))
```

NEW

PURPOSE

NEW saves the values of the specified local variables. The values are restored at the next QUIT statement.

FORMS:

N{EW}{:*postcond*}

N{EW}{:*postcond*}(SP)*argument*

In which *argument* can be one of the following:

name

(*local*{,...})

@expr atom

where:

postcond	is a postconditional expression
name	is the name of a local variable without subscripting
local	is the name of a local variable or an indirect reference that evaluates to a local variable name or to another indirect reference
@expr atom	is an indirect reference that evaluates to one or more NEW arguments

EXPLANATION:

The NEW command allows you to save the values of variables temporarily. Variables specified in the NEW command are saved by being placed in a stack; the specified variables are then deleted (killed) from the local symbol table. When a QUIT is reached, the stacked values for the variables are restored to the symbol table. The QUIT can be either explicit or implicit (but not within the scope of a FOR loop).

Using the NEW command is most useful when you have local variables in a subroutine that may conflict with local variables in the routine that calls the subroutine. Using NEW allows you to assign new values to these variables while preserving the old values.

NEW (Cont.)

When you use the NEW command, the value of the specified variables are undefined until they are assigned new values. It is as though the variables had been killed (except that the values are restored at a later time). If you apply $DATA to the variable you get a value of zero (unless a new value has been assigned to the variable).

When a QUIT is reached, the new values assigned to the specified variables are deleted (the variables are killed); and the old (stacked) values are assigned to the variables.

NEW without arguments stacks all of the variables in the local symbol table. The local symbol table is then empty.

The effect of NEW with arguments depends on the argument form that you use.

1. *local*

   This form of the NEW command is selective. The only variable that is stacked is the one that you specify.

   If the variable you specify is subscripted, then the NEW command stacks the variable and all of its descendant nodes. For example, you have an array containing the nodes:

   X(1,2)
   X(1,2,1)
   X(1,2,3)
   X(2,2)

   If you specify X(1,2) as the NEW argument, then X(1,2), X(1,2,1), and X(1,2,3) are all stacked; but X(2,2) is not stacked.

2. (*local*{,...}

   This is the exclusive form of NEW. All variables in the local symbol table are stacked except those that are listed in the argument. The variables in the argument are preserved in the symbol table. All other variables are deleted from the symbol table.

COMMENTS:

Keep the following points in mind when you use the NEW command:

1. The NEW command works only on local variables, not global variables.

2. Use of the NEW command causes partition space to be used for saving variables in a push-pop stack. Excessive use of the NEW command can exhaust all of the partition space, causing an error.

3. In Programmer Direct Mode, you can use NEW to stack variables, but the effect is the same as a KILL. The variables can never be restored since no QUIT can ever be done from this level to a lower level (you are at the lowest level already).

4. Error processing restores variables stacked with NEW.

5. Using CTRL/Y causes the stack of variables established with NEW to be erased. These values are lost.

RELATED:

The QUIT Command
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example saves (stacks) the variable X and any descendants it may have.

```
NEW X
```

The following example stacks all variables in the local symbol table.

```
NEW
```

Consider a local symbol table with the following variables:

A = 1
B = 2
C = 4
X = 6

The following example preserves A and B in the symbol table:

```
NEW (A,B)
```

The new symbol table is:

A = 1
B = 2

The variables that are stacked are:

C = 4
X = 6

The following example uses a postconditional expression. If X = 1, then NEW stacks the variable Y.

```
NEW:X=1 Y
```

OPEN

PURPOSE:

OPEN obtains ownership of one or more devices.

FORM:

O{PEN}{:*postcond*}(SP)*argument*,...

In which *argument* can be one of the following:

device{:{*params*}:*timeout*}

@expr atom

where:

postcond	is a postconditional expression
device	is a device specifier
params	are one or more expressions that evaluate to device opening parameters
timeout	is a timeout
@expr atom	is an indirect reference that evaluates to one or more OPEN arguments

EXPLANATION:

OPEN reserves the device(s) specified in the argument(s) for your use. For any device except the principal device, you must issue an OPEN command to reserve the device before you issue a USE command to direct all input and output operations to it. (For an exception to this rule, see the ZUSE command.)

The argument(s) for OPEN consists of one or more valid device specifier(s) and their modifying parameters. The specifier(s) can be by argument or name indirection.

If you do not include modifying parameters, DSM-11 uses default system parameters. If you do include modifying parameters, DSM-11 uses the parameters as specifications of action to take when it opens the device. The effects of these parameters can last until you issue a CLOSE command to close the device. If you do not relinquish device ownership with CLOSE, DSM-11 ends that ownership and the use of any modifying parameters you specify when you execute a HALT.

OPEN (Cont.)

See the *DSM-11 User's Guide* for a description of device specifications and the parameters you can use with them.

COMMENTS:

Keep the following points in mind when you use the OPEN command:

1. Two users cannot own the same device. If you attempt to open a device already owned by another user, DSM-11 suspends execution of your job until the device is free.

   To avoid a potential suspension of execution, use a timeout. If DSM-11 cannot open the device during the timeout, it sets $TEST to zero and resumes execution. If DSM-11 can open the device during the timeout, it opens the device, sets $TEST to true (one), and resumes execution.

   DSM-11 resumes execution as soon as it can open the device. It ignores any time remaining on the timeout.

2. Opening a device does not make it your current input/output device; it only gives you ownership of the device. You must execute a USE statement to make the device your current input/output device.

3. If you specify more than one parameter with a device, you must enclose the parameters in parentheses and separate the parameters with colons.

RELATED:

The CLOSE Command
The HALT Command
The USE Command
The ZUSE Command
The $IO Special Variable
The $TEST Special Variable
Postconditional Expressions (Section 2.7.4)
Timeout Expressions (Section 2.7.5)

EXAMPLES:

The following example obtains ownership of device 16. If another user already owns device 16, execution hangs until the device is available.

```
O 16
```

The following example shows the format of the OPEN command with a timeout if no parameters are present.

```
OPN   O 16::10
```

OPEN (Cont.)

The following example prompts for a device; then, it tries to open the device for up to three seconds. If it is successful, DSM-11 sets $TEST to 1. Otherwise, DSM-11 sets $TEST to 0 and continues execution at the line labeled REGO.

```
C       W !,"ENTER DEVICE SPEC:" R !,DEV
        O DEV::3 G:'$T REGO
```

The following example writes the first subscript level from the global ^NAM into Sequential Disk Processor (SDP) space. The parameters with the OPEN command are the starting byte and block address.

```
GO      O 59:(0:4402:"DL0") U 59 S E=""
        S E=$O(^NAM(E)) G:E="" STOP
        W E," " W:$D(^NAM(E))#10 ^NAM(E),! G GO+1
STOP    W "STOP",! S X=$ZA
        C 59 U 0 W X
```

4. The QUIT on line B returns control to the statement immediately following "D B" on line A.

COMMENTS:

Keep the following points in mind when you use QUIT:

1. If you use a postconditional expression with QUIT, DSM-11 executes the QUIT only if the postconditional expression has a value of true.

2. If you do not include a QUIT in a routine, DSM-11 terminates the routine after it executes the last sequential line unless it encounters a statement redirecting control elsewhere. In block-structured programming, DSM-11 terminates an execution block when it reaches a routine line of a lower level than that of the execution block. See Section 1.7 for more discussion of block-structured programming.

3. When DSM-11 encounters a QUIT in an error handling routine (declared as an error-handler in the $ZTRAP special variable), it returns control to the point immediately following the DO argument in which the routine that caused the error was called.

4. If any local variables have been saved with the NEW command, the next QUIT that is not within the scope of a FOR loop restores these variables.

RELATED:

The DO Command
The FOR Command
The HALT Command
The NEW Command
The XECUTE Command
Extensions to Routine Structure (Section 1.7)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example executes a QUIT if XR equals zero.

```
COND   Q:'XR
```

The following example routine line keeps a total of numbers entered in response to the prompt. The routine line quits when the value of A is null.

```
Z      S SUM=0 F I=0:1 R !,"*",A Q:A="" S SUM=SUM+A
```

READ

PURPOSE:

READ receives data from the current device.

FORMS:

R{EAD}{*postcond*}SP*argument*,...

In which *argument* can be one of the following:

format

string

local{:*timeout*}

local{#*inp field*}{:*timeout*}

**local*{:*timeout*}

@expr atom

where:

postcond	is a postconditional expression
format	is one or more formatting characters
string	is a string literal
local	is the name of a local variable with or without subscripting
timeout	is a timeout
inp field	is an integer-valued expression preceded by the pound sign (#) that specifies the number of characters to be read
@expr atom	is an indirect reference that evaluates to one or more READ arguments

EXPLANATION:

The action READ performs depends on the argument form you use, as follows:

1. *format*

READ (Cont.)

READ performs the operations specified by the formatting character(s) in the argument. DSM-11 updates the special variables $X and $Y to reflect the result of the operation.

If the device does not accept output, DSM-11 ignores the formatting characters. If the device does not accept input, DSM-11 reports an error.

See the reference descriptions in Chapter 6 for more information on $X and $Y.

2. *string*

READ writes the string specified on the current device. DSM-11 updates the $X special variable to reflect the length of string.

If the device does not accept output, DSM-11 ignores the string but does update $X to reflect the change. If the device does not accept input, DSM-11 reports an error.

3. *local*{:*timeout*}

DSM-11 reads a string (of up to 255 characters) from the current device and assigns the value of that string to the local variable. It also updates $X to reflect the data received.

A read terminates when one of the following occurs:

- You type a line terminator (such as a RET)
- A timeout occurs
- The string of characters you are typing reaches the field length

If a terminal is the current device, you can use a timeout to limit the amount of time DSM-11 waits for the string. (If you do not use a timeout, DSM-11 suspends execution until it receives an input string followed by a carriage return or other valid line terminator.)

If DSM-11 receives data followed by a line terminator before the timeout occurs, it assigns the value of the data to the local variable, updates $X, and assigns a value of true (one) to the $TEST special variable. DSM-11 then ignores any time remaining in the timeout and resumes execution.

When a timeout occurs while you are entering data, DSM-11 terminates the data with a line terminator. The value of the data that DSM-11 has received is assigned to the local variable. DSM-11 does not know that this data is incomplete from your standpoint. DSM-11 also assigns a value of false (zero) to $TEST and updates $X to reflect the input string.

If DSM-11 does not receive any data before the timeout occurs, it assigns a null string ("") as the value of the local variable. It also assigns a value of false (zero) to $TEST and updates $X to reflect the input string.

If you exceed the field length (default of 255) when typing in data, DSM-11 reads the data through the full field length (255 characters) and then terminates the read with a line terminator. The value of the data that DSM-11 has received is assigned to the local variable.

4. *local*{*#inp field*}{*:timeout*}

DSM-11 reads a string from the current device whose length is determined by the value of *inp field* and assigns the string to a local variable. *Inp field* must be an integer-valued expression from 1 to 255.

If you include a timeout after the input field specification, it affects the READ operation the same as the *local*{*:timeout*} argument, described above.

If you exceed the field length when typing in data, it affects the read in the same way as described for the *local*{*:timeout*} argument, except that the field length is the one that you specify, not 255.

READ operations with an input field specifier also affect the $X and $TEST special variables in the same way as the *local*{*:timeout*} argument.

5. **local*{*:timeout*}

DSM-11 accepts as input the next character received. DSM-11 then assigns the decimal equivalent of the ASCII code for the character as the value of the local variable.

If a terminal is your current device, you can also use a timeout with this form to limit the amount of time DSM-11 waits for data. (If you do not use a timeout, DSM-11 waits until it receives a character. The character does not have to be followed by a carriage return or other valid line terminator.)

If DSM-11 does receive the single-character before the timeout occurs, it assigns the decimal ASCII code of that character as the value of the local variable and assigns a value of true (one) to $TEST. DSM-11 then ignores any time remaining in the timeout and resumes execution.

If DSM-11 does not receive the single-character input before the timeout occurs, it assigns a value of -1 to the local variable. It also assigns a value of false (zero) to $TEST.

In neither case does DSM-11 update $X or $Y. The READ * form does not affect $X or $Y.

COMMENTS:

Keep the following points in mind when you use the READ command:

1. With terminals, DSM-11 has a type-ahead feature that allows you to enter data before DSM-11 executes the READ argument that assigns that data as the value of a local variable. DSM-11 stores the type-ahead data in an input buffer and retrieves the data when it executes the READ argument.

   However, if you use a READ command in which you precede the local variable with a format character or a READ command with a string-literal output as a prompt for input, DSM-11 clears the input buffer, discarding all previously typed data.

   To keep the type-ahead feature and provide interactive queries and responses in application programming, do all formatting and prompting with WRITE statements. For example, you can rewrite the statement:

   ```
   R !!,"CUSTOMER NAME? ",!,NAM
   ```

   as:

   ```
   W !!,"CUSTOMER NAME? ",! R NAM
   ```

2. If *inp field* is zero, DSM-11 returns an error.

3. If the input field length has been specified with a previous USE command input buffer parameter, the READ argument takes precedence, and DSM-11 reads *inp field* characters.

4. If *inp field* has an integer and fractional component, DSM-11 discards the fraction and reads the integer number of characters. Thus, the statements:

   ```
   R X#5.5
   HELLOO
   ```

   assign X the value:

   HELLO

RELATED:

The WRITE Command
The $ASCII Function
The $CHAR Function

READ (Cont.)

The $TEST Special Variable
The $X Special Variable
The $Y Special Variable
Formatting Characters (Section 2.6.7)
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)
Timeout Expressions (Section 2.7.5)

EXAMPLES:

The following example writes the string MESSAGE on the current I/O device.

```
R "MESSAGE"
MESSAGE
```

The following example causes four new-line operations.

```
JMP    R !!!!
```

The following example prompts for a number and waits ten seconds for a response. If it receives no data, DSM-11 transfers control to the line labeled A3.

```
Z      R !,"NUMBER: ",N:10
       G:'$T A3
```

The following example waits ten seconds for a number. If it receives the number, it assigns the variable N the value of the number and sets $TEST to one. If it does not receive the number, it sets $TEST to zero.

```
A      S B=10,MESS="ENTER A NUMBER"



B      W !,MESS R N:B
```

The following example reads the first four characters entered and writes them to the principal device.

```
U 0
R X#4 W !,X
TESTING
TEST
```

The following example sets the size of the READ input buffer depending on the value of a variable.

READ (Cont.)

```
STRT   R "Enter Number: ",VAL
       I VAL)99 S IBUF=3
       E  S IBUF=2
       R !,"Enter Next Number: ",NVAL#IBUF
```

The following example requests a single-character input within a 5-second interval. If DSM-11 receives a character in that time, it sets A to the ASCII decimal code for that character and sets $TEST to one.

```
RD     R *A:5
```

SET

PURPOSE:

The SET command assigns the value of an expression to a variable or to a substring within a variable.

FORM:

S{ET}{:*postcond*}␣*argument*,...

In which *argument* can be one of the following:

storage ref = *expression*

(*storage ref*{,...}) = *expression*

piece ref = *expression*

@expr atom

where:

postcond	is a postconditional expression
storage ref	is a local or global variable name (defined or undefined) with or without subscripts
expression	is an expression
piece ref	is a reference to one or more substrings within a local or global variable (defined or undefined) in the syntax of the $PIECE function
@expr atom	is an indirect reference that evaluates to one or more SET arguments

EXPLANATION:

DSM-11 evaluates the arguments of the SET command in the following order:

1. Occurrences of indirection or subscripts to the left of the equal sign are evaluated in left-to-right order to determine the storage or piece reference.

2. The expression to the right of the equal sign is evaluated.

3. The expression to the right of the equal sign is assigned to the storage reference(s) or to the piece reference(s) to the left of the equal sign.

SET (Cont.)

The specific action that SET performs depends on the argument form you use, as follows:

1. *storage ref* = *expression*

   SET assigns the value of an expression to a storage reference.

2. (*storage ref*{,...}) = *expression*

   SET assigns the value of an expression to all named storage references.

3. *piece ref* = *expression*

   SET assigns the value of an expression to one or more substrings within the named storage reference. DSM-11 can store data as a string composed of a series of substrings separated by a delimiter. The delimiter can be any character or sequence of characters that occur in the string. When a string consists of substrings separated by a delimiter, all substrings within that string have a position with respect to the delimiter. For example, consider the following string:

   "ABC;DEF;GHI"

   If the semicolon (;) is the delimiter, the characters ABC form the first substring; DEF form the second substring; and GHI form the third substring.

   The *piece ref* argument uses a syntax that resembles the DSM-11 $PIECE function to specify the substring(s) to be set within a variable. This syntax is:

   $P{*IECE*}(*storage ref,delimiter*){,*start field*}{,*end field*})

   where:

   $P{*IECE*} is the *piece ref* prefix

   storage ref is a storage reference that contains the substring that you want to assign a value to

   delimiter is the character or characters used as a delimiter between the substrings in *storage ref*

   start field is an integer expression that specifies the substring (or first of a series of substrings) that you want to assign a value to

   end field is an integer expression that specifies the last in a series of substrings that you want to assign values to

In the two-argument form of *piece ref*, SET assigns a value to the first substring in *storage ref*. If the value of FST is:

"HELLO$WORLD"

the statement:

```
S $P(FST,"$")="GOODBYE"
```

assigns the value of the first substring in FST equal to GOODBYE, as shown below:

```
W FST
GOODBYE$WORLD
```

In the three-argument form of *piece ref*, SET assigns a value to the substring specified by *start field*. If the value of X is:

"129 GRASMERE STR;BRIGHTON;MASSACHUSETTS"

the statement:

```
S $P(X,";",2)="NEWTON"
```

assigns the value of the second substring in X equal to NEWTON, as shown below:

```
W X
129 GRASMERE STR;NEWTON;MASSACHUSETTS
```

In the four-argument form of *piece ref*, SET assigns values to the substrings from *start field* to *end field* inclusive. If the value of ALPHA is:

"ABC;DEF;123;456;789;PQR"

the statement:

```
S $P(ALPHA,";",3,5)="GHI;JKL;MNO"
```

assigns ALPHA substrings three, four, and five the values GHI, JKL, and MNO respectively, as shown below:

```
W ALPHA
ABC;DEF;GHI;JKL;MNO;PQR
```

COMMENTS:

Keep the following points in mind when you use the SET command.

SET (Cont.)

For all forms:

1. The evaluation order in the SET command can affect how DSM-11 determines naked references. Consider the following example:

```
S ^X(3)=3,^(2)=^Z(5)
```

   First, DSM-11 sets ^X(3) equal to three and sets the naked indicator by the last full global reference, ^X(3). (The naked indicator is at the first level.)

   However, when DSM-11 evaluates the expression to the right of the equals sign in the second argument, it finds a new global, ^Z, to serve as the value of the naked indicator. Therefore, it determines what the storage reference on the left of the argument (^(2)) is by using ^Z(5).

   Thus, DSM-11 evaluates the second argument in this SET command as:

```
S ^Z(2)=^Z(5)
```

2. If you use a postconditional expression with the SET command, DSM-11 executes the SET only if the postconditional has a value of true.

3. You cannot use a postconditional with a SET argument. However you can use the $SELECT function to conditionally assign a value to a storage reference or a storage reference to a value. For example, the statement:

```
S A=$S(J>100:M,1:N)
```

   assigns A the value of M if J is greater than 100 and assigns A the value of N in all other cases.

   The statement:

```
S @$S(A:"A",1:"B")=0
```

   assigns A the value of zero if A is not already zero or assigns B the value of zero if A is already zero.

4. You cannot use SET to assign a value to any DSM-11 special variables except $ZBREAK, $ZERROR, and $ZTRAP. $ZBREAK is used to debug routines. $ZBREAK is used to debug routines. $ZERROR and $ZTRAP are used in error processing.

See the description of $ZBREAK, $ZERROR, and $ZTRAP in Chapter 7 for details about these special variables.

For the *piece ref*=*expression* form:

1. The two-argument form of *piece ref* is equivalent to the three-argument form when *start field* is one, as shown in the following example:

```
S (STR1,STR2)="NO#WAY#JACK"
S $P(STR1,"#")="MAKE" S $P(STR2,"#",1)="MAKE" W !,STR1,!,STR2

MAKE#WAY#JACK
MAKE#WAY#JACK
```

2. If *storage ref* consists of fewer than the specified number of substrings, DSM-11 assigns *storage ref* the current value of *storage ref* concatenated with the number of *delimiter* characters needed to bring the total number of delimiters to *start field*-1, concatenated with *expression*.

   Thus, if the value of MONTHS is:

   "JAN;FEB;MAR;APR"

   the statement:

```
S $P(MONTHS,";",8)="AUG"
```

   assigns MONTHS the value:

```
"JAN;FEB;MAR;APR;;;;AUG"
```

3. If *storage ref* is undefined, DSM-11 assigns *storage ref* the value of *expression* preceded by *start field*-1 *delimiter* characters.

   Thus, if the variable TST does not exist ($DATA(TST)=0), the statement:

```
S $P(TST,";",7)="123"
```

   assigns TST the value:

   ;;;;;;123

4. If a four-argument *piece ref* specifies more substrings than contained within the expression to the right of the equal sign, DSM-11 assigns

the specified range of substrings the value of *expression*. Thus, if the value of VAR is:

"ABC;DEF;GHI;JKL;MNO"

the statement:

```
S $P(VAR,";",2,4)="123;456"
```

assigns VAR substrings two, three, and four the value 123;456 (effectively reducing the total number of substrings in VAR to four). The resulting value for VAR is:

"ABC;123;456;MNO"

RELATED:

The Binary EQUALS Operator
The $PIECE Function
The $SELECT Function
The $ZBREAK Special Variable
The $ZERROR Special Variable
The $ZTRAP Special Variable
Global Variables (Section 2.4.2)
Indirection Operator (Section 2.6.6)
Local Variables (Section 2.4.1)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example sets the variable XYZ to 42465.

```
S XYZ=42465
```

The following example sets several variables and writes the result.

```
S I=3,D=I+5,C=I*D W C
24
```

The following example assigns the value of zero to the variables X,Y,and Z.

```
S (X,Y,Z)=0
```

The following example determines the naked reference by the previous global reference ^C(2) --- not ^A(2) which DSM-11 evaluates later. Note that the later assignment of an expression to a variable (^A(2)) cancels any previous assignment to that variable.

```
S ^A(2)=1,^C(2)=2,^A(2)=^(2)+1
W ^A(2),!,^C(2)
3
2
```

The following example uses a postconditional. If the variable LM has a value of 4, the example gives it the value zero.

```
S:LM=4 LM=0
```

The following examples use name and argument indirection to assign the local variable A1 the value of one.

```
S A="A1"
S @A=1

S A="A1=1"
S @A
```

The following example stores data in a global that uses a person's name as an identifying subscript. The routine stores the data for each person as a series of substrings separated by a semicolon (;), starting with the person's address. Note that the piece-by-piece construction of the global occurs only after the name is stored as part of the full global reference.

```
STRT   S PMT(1)="NAME: ",PMT(2)="ADDRESS: ",PMT(3)="CITY: "
       S PMT(4)="STATE: ",PMT(5)="ZIP CODE: "
       S PMT(6)="PHONE: ",PMT(7)="SOCIAL SECURITY NO: "
       F I=1:1:7 W PMT(I) R INP(I) S:I)1 $P(^DEMOG(INP(1)),";",I-1)=INP(I)
```

The following example sets the naked indicator, then assigns a value to the second piece of the next naked reference. The example assigns the value by determining the value of the fourth piece of the following naked reference (using the $PIECE function to determine that value). The example assumes that ^X(3) is undefined and ^X(4) is defined to be the string "THIS#IS#A#TEST".

```
S ^X(2)=1
S $P(^(3),";",2)= $P(^(4),"#",4)
W !,^X(3),!,^X(4)

TEST

THIS#IS#A#TEST
```

USE

PURPOSE:

USE makes the device specified in its argument the current I/O device. Unless the device is the principal device, you must have previously gained ownership of it with an OPEN statement.

FORM:

U{SE}{:*postcond*}(SP)*argument*

In which *argument* can be one of the following:

device{:*params*}

@expr atom

where:

postcond	is a postconditional expression
device	is a device specifier
params	are one or more expressions that specify device parameters
@expr atom	is an indirect reference that evaluates to a USE argument

EXPLANATION:

DSM-11 makes the device you specify the current device. It performs all input/output operations with that device until one of the following situations occurs:

1. You close the device with CLOSE --- your principal device then becomes the current I/O device.
2. You execute another USE to select a new device.
3. An error occurs when you have not set $ZTRAP --- DSM-11 then resets your principal device as your current device and sends the error message to it.
4. You execute a HALT command.

The arguments for USE consist of a valid device specifier and its optional, modifying parameters. If you enter more than one device specifier in the argument list, DSM-11 uses the last (rightmost) device you specify as the current I/O device.

USE (Cont.)

If you do not include modifying parameters, DSM-11 uses the system defaults. If you include modifying parameters, DSM-11 uses the parameters as specifications of action to take when it directs I/O operations to the specified device. The effect of these parameters can last until you direct I/O to another device.

See the *DSM-11 User's Guide* for a description of device specifications and the parameters you can use with them.

COMMENTS:

Keep the following points in mind when you use the USE command:

1. You cannot issue a valid USE command for a device you do not own. USE only directs I/O to an already owned device. Thus, you must gain ownership of any device (other than your principal device) with an OPEN command before you can use it.

2. If you specify more than one parameter with a device, you must enclose the parameters in parentheses and separate the parameters with colons.

3. The $IO special variable contains the specification of the device you are using for input and output. Thus, $IO has the value of either the principal or the current device.

   The principal device is the device the system opens when you log into the system and uses by default for input and output. You can refer to the principal device as device 0.

   The current device is any device you open with an OPEN command and use for input and output with a USE command. Thus, the current device is the device you are presently using for input and output operations.

   4. The $X and $Y special variables always reflect the state of the device you are currently using. When you are using a device, DSM-11 writes any horizontal or vertical cursor or carriage specifications associated with the device into the $X and $Y special variables.

      DSM-11 retains the old values for any device you previously used. If you use that device again, the values in $X and $Y are the values for that device.

   5. The $ZA and $ZB special variables can also reflect the state of the device you are currently using. DSM-11 also retains the values of $ZA and $ZB for any device you previously used. If you use that device again, the values in $ZA and $ZB are the values for that device.

   RELATED:

USE (Cont.)

The CLOSE Command
The OPEN Command
The $IO Special Variable
The $X Special Variable
The $Y Special Variable
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example specifies device 3 as the current I/O device.

```
U 3
```

The following example uses argument indirection to use the sequential disk processor. (See the *DSM-11 User's Guide* for more information on the sequential disk processor.)

```
B   S A="59:(0:2369300)"
    U @A
```

VIEW

PURPOSE:

VIEW allows you to read and write blocks to disk storage or to alter locations in memory.

RESTRICTION:

You can only use the VIEW command if you are using the system UCI, are running a library routine that contains VIEW statements, or are working on a system that has been configured to allow unrestricted use of VIEW.

See your System Manager to determine whether VIEW is restricted.

FORMS:

V{IEW}␣*argument,...*

In which argument can be one of the following:

{-}*block addr:disk*

{-}*block addr:volume set*

{-}*block addr*

destination:{*state*}:*value*

destination:{*state*}:*source*:{*state*}:*extent*

@expr atom

where:

block addr	is an integer expression that specifies the address of a disk block
disk	is an alphanumeric expression that specifies a disk drive
volume set	is an alphanumeric expression that specifies a mounted volume set
destination	is an integer expression that specifies a memory location
state	is an integer expression that specifies a state (see Table 4-1)
value	is an integer expression that specifies the value to write to the location

VIEW (Cont.)

source	is an integer expression that specifies a memory location
extent	is an integer expression that specifies the number of bytes to be transferred
@expr atom	is an indirect reference that evaluates to one or more VIEW arguments

EXPLANATION:

The function VIEW performs depends on the argument form you use:

1. {-}*block addr*:*disk* or {-}*block addr*:*volume set* or {-}*block addr*

   VIEW allows you to examine the contents of a disk block. DSM-11 uses a buffer in memory called the VIEW device for all VIEW transfers to and from a disk block. Before you can use VIEW to access from disk, you must gain ownership of the VIEW device (device 63) with an OPEN statement.

   (See the *DSM-11 User's Guide* for a description of the OPEN and USE parameters for the VIEW device.)

   After you gain ownership of the VIEW device, enter a VIEW command with an argument specifying the block to be read in either of the three following forms:

   V *block addr*:*disk*

   V *block addr*:*volume set*

   DSM-11 then reads the specified block into the VIEW device.

   To access the block, use:

   a. VIEW with a destination:0:value argument to write a new value into the VIEW device

   b. The $VIEW function to read the value in the VIEW device

   Always use a *state* value of 0 when you access a value in the VIEW device.

   After you finish with the block, write it back to disk with an argument consisting of the block address preceded by a Unary MINUS, using either of the two following forms:

   V -*block addr*:*disk*

V *-block addr:volume set*

The disk that the block is on is specified by a three-letter mnemonic code such as DM0, indicating drive 0 of an RK06 or RK07 drive. In this way you can access the block address by the relative block number for that disk.

The following is a list of disk mnemonics:

Mnemonic	Type	Type Designation
DK	RK05	0
DM	RK06, RK07	1
DR	RM02, RM03, RM05	2
DB	RP04, RP05, RP06	3
DL	RL01, RL02	4
DU	RA80, RA60, RA81, RD51, RX50	5

You can also specify the block address within a volume set by specifying the block number relative to the beginning of the volume set. In this case *disk* is replaced by a code such as S0 or S1, indicating, for example, Volume Set 0 or Volume Set 1. Volume Set 0 is always the system volume set, which contains the system disk.

In a volume set, block numbering proceeds sequentially from the beginning of the volume set and may proceed sequentially from one disk to another if there are several disks within the volume set.

You can use the Volume Set Table (described in the *DSM-11 User's Guide*) to determine where a block is within a given volume set.

If necessary (though not recommended), you can also specify a block address by calculating the DSM-11 block number. In this case the syntax is one of the following:

V *block addr*

V *-block addr*

You must not specify a *disk* or *volume set*. The DSM-11 block number or *block addr* is calculated as follows:

disk type*2097152 + (disk unit number*262144) + absolute block number

This syntax is not recommended, however, because of a limit on the absolute block number of 262,144. The syntax is included here for compatibility with previous DSM-11 releases. See Appendix D of the

DSM-11 User's Guide for information on how to calculate the absolute block number.

2. *destination*{:*state*}:*value*

   VIEW writes to either the VIEW device or to memory. To write to the VIEW device, you must first open device 63.

   The first argument element, *destination*, specifies a word location in which you want to write. Because DSM-11 only writes full words to memory, *destination* must be an even value.

   The second argument element, *state*, indicates the meaning of *destination* and the type of operation you want to perform. If you do not specify a *state* value, VIEW considers that *destination* is a logical address within Kernel-Mode mapping (*state* value of -1).

   The third argument element, *value*, is new value you want to write to the location indicated by *destination*.

   Table 4-1 describes the *state* values, their meanings, and the appropriate *destination* values to use with them.

Table 4-1: VIEW Argument Values

destination or *source* Value	*state* Value	Meaning
0 --- m	1 --- n	*state* specifies a job number. (n stands for the maximum number of jobs on the system.) *destination* or *source* specifies an offset from the beginning of the partition. (m stands for the largest offset value possible in the partition.) To specify your own job, use the special variable $JOB for *state*.
0 --- x	0	*state* specifies that *destination* or *source* is an offset from the beginning of the VIEW Buffer. (x stands for the largest offset value on the VIEW device.)
0 --- 65534	-1	*state* specifies that *destination* or *source* is the logical address used in Kernel-Mode mapping. (This is the default *state* when you do not include a *state* specifier.)
0 --- 65534	-2	*state* specifies that *destination* or *source* is the logical address used in User-Mode mapping.
0 --- 16382	+128 to +32767	*state* specifies a memory-management block.*destination* or *source* is an offset starting at the memory-management block specified by *state*.

When the *state* value is either -1 or -2, a *destination* or *source* of 40960 (words) through 56344 (words) represents the current partition. A *destination* or *source* value of 56344 (words) to 65535 (words) represents the I/O page.

When you use a *state* value between +128 and +32767, you can view an area in memory that would otherwise not be mapped by either Kernel Mode (*state* = -1) or User Mode (*state* = -2) at the point in time when you execute the VIEW. Thus, these *state* values allow you to view anywhere in physical memory.

To obtain a more detailed explanation, see your *PDP-11 Processor Handbook* for information on memory management.

3. *destination:*{*state*}:*source*:{*state* }:*extent*

   The five-argument form of the view command is similar to the three-argument form. The five-argument form allows you to move a number of bytes from one memory area to another.

   The values for the *state* argument are the same as those in Table 4-1. The *state* is optional and defaults to a value of -1 if no value is present. If you do use the default value, you must include the correct number of colons to distinguish this form from the three-argument form:

   destination::*source*::*extent*

   The first two arguments, *destination* and *state*, are similar to the arguments for the three-argument form. The *destination* value must be an even number and is the address to which you are moving the bytes. The *state* value indicates the meaning of *destination* and the type of operation.

   The third argument, *source*, is the location from which you are moving bytes; it must be an even number. The fourth argument, *state*, indicates the meaning of *source* and the type of operation.

   The fifth argument, *extent*, indicates the number of bytes that are to be moved. It must be an even number.

RELATED:

The $VIEW Function
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example finds a particular device (DEV) in the device table (DEVT), initializes the low byte of the entry, and adds the value of JOBE to the entry.

```
V DEVT+DEV::$V(DEVT+DEV)256*256+JOBE
```

The following example opens the VIEW Buffer and initializes a map block. (See the *DSM User's Guide* for more information on map blocks.)

VIEW (Cont.)

```
STA   0 63 V MBLK
      F I=0:2:796 V I:0:0
      V 796:0:65535
      V 1006:0:65535,1008:0:21845,1010:0:43690
      V 1012:0:32769
      V 1022:0:399
      V -MBLK
```

The following shows how to read a disk-relative block into the VIEW buffer from a disk:

```
V 2214:"DK1"
```

The following shows how to read a disk-relative block into the VIEW buffer from a volume set:

```
V 5213:"S0"
```

WRITE

PURPOSE:

WRITE sends data and/or control information to the current device.

FORM:

W{RITE}{:*postcond*}SP*argument*,...

In which *argument* can be one of the following:

format

expression

**integer*

@expr atom

where:

postcond	is a postconditional expression
format	is one (or more) formatting characters
expression	is an expression
integer	is an integer expression whose value is the decimal equivalent of an ASCII character value
@expr atom	is an indirect reference that evaluates to one or more WRITE arguments

EXPLANATION:

The action WRITE performs depends on the argument form you use.

1. *format*

   WRITE performs the formatting operation specified by the formatting characters in the argument. DSM-11 updates the special variables $X and $Y to reflect the change.

2. *expression*

WRITE writes the specified expression on the current device. DSM-11 adds the length of the expression to $X.

If the current device does not accept output, DSM-11 performs no output but does add the length of the string to $X.

3. **integer*

   This form is called the WRITE * (WRITE star) form. WRITE star writes the ASCII character whose decimal code is equivalent to the integer expression you use as an argument.

   If the ASCII character is a graphic character, DSM-11 writes the character on the current device. If the ASCII character is a control character, DSM-11 performs the operation specified by the control character on the current device.

   In neither case does DSM-11 change the values in $X and $Y. The WRITE * form does not affect $X and $Y.

COMMENTS:

Keep the following points in mind when using the WRITE command:

1. The WRITE * forms --- WRITE *12 (form feed) and WRITE *13 (new-line operation) --- are not equivalent to WRITE # AND WRITE !. To perform formatted writes, you should use the formatting characters # (form feed) and ! (carriage return/line feed).

2. When you issue a WRITE statement to a terminal or line printer, start or end each output line with a carriage return/line feed formatting character (!) to prevent overprinting on the device.

   When you use the line printer, enter a carriage return/line feed formatting character after the last item you want to print and before you close the device. Otherwise, you can lose the last line.

RELATED:

The READ Command
The $CHAR Function
The $X Special Variable
The $Y Special Variable
Formatting Characters (Section 2.6.7)
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

WRITE (Cont.)

The following example writes the result of executing the $DATA function on the local node L(1,3).

```
S L(1,3)="123.77"
W $D(L(1,3))
1
```

The following example performs four new-line operations on the current device.

```
W !!!!
```

The following example prints a string literal in quotes on the current device. (To print a string literal in quotation marks, you must include two additional sets of double quotes within the mandatory pair.)

```
W """THIS IS A QUOTED STRING"""
"THIS IS A QUOTED STRING"
```

The following example skips three lines and displays the expression represented by M if V is greater than five.

```
B2 W:V>5 !!!,@M
```

The following examples ring the bell on a terminal (if a terminal is the current device and if that terminal has a bell).

```
W *7
```

or:

```
W $C(7)
```

The following example uses argument indirection to perform one form feed operation, two new-line operations and a horizontal tabulation of 20 spaces from the left margin. Then it writes the string literal TEST.

```
J1   S A="#!!?20"

     .
     W @A,"TEST"
```

XECUTE

PURPOSE:

XECUTE can execute DSM-11 statements that result from the process of expression evaluation.

FORM:

X{ECUTE}{:*postcond*}␣*argument*,...

In which *argument* can be one of the following:

expression:{*postcond*}

@expr atom

where:

postcond	is a postconditional expression
expression	is an expression that evaluates to one or more DSM-11 statements
@expr atom	is an indirect reference that evaluates to one or more XECUTE arguments

EXPLANATION:

Each XECUTE argument must evaluate to a string containing DSM-11 statements. The string should not contain a TAB character at the beginning or a carriage return at the end.

In effect, each XECUTE argument is like a one-line subroutine called by a DO command and terminated with a QUIT command. After DSM-11 executes the argument, it returns control to the point immediately following the XECUTE argument.

If an XECUTE argument contains a FOR command, the implicit QUIT is beyond the scope of the FOR. That is, the XECUTE argument behaves as if it were a two-line subroutine with the second line containing only a QUIT statement.

If an XECUTE statement contains a DO command, DSM-11 executes the routines specified in the DO argument(s). When it encounters a QUIT, it returns control to the point immediately following the DO argument.

XECUTE (Cont.)

For example, in the following statement, DSM-11 executes the routine ROUT and returns to the point immediately following the DO argument to write the string DONE.

```
X "D ^ROUT W !,""DONE"""
```

If an XECUTE argument contains a GOTO command, DSM-11 transfers control to the point specified in the GOTO argument. When it encounters a QUIT, it returns control to the point immediately following the XECUTE argument that contained the GOTO statement.

For example, in the following statement, DSM-11 transfers control to the routine ROUT and returns to write the string FINISH. It never writes the string DONE.

```
X "G ^ROUT W !,""DONE""" W !,"FINISH"
```

COMMENTS:

You can use postconditional expressions with both the XECUTE command and its argument. DSM-11 evaluates an XECUTE with a postconditional expression only if that postconditional expression has a value of true. DSM-11 executes an XECUTE argument with a postconditional expression only if the postconditional expression has a value of true.

RELATED:

The DO Command
The FOR Command
The GOTO Command
The QUIT Command
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example sets X to four if L equals two. Because of the IF statement in the XECUTE argument, DSM-11 also sets the $TEST special variable.

```
S M="I L=2 S X=4" X M
```

The following example uses XECUTE to create an "if-then-else"construction.

```
X "S A=$S(A)100:B,1:D)"
```

The following example executes the subroutine that is the value of A.

XECUTE (Cont.)

```
S A="W ! F I=1:1:5 W ?5,I+1"
X A
    2    3    4    5    6
```

Chapter 5
DSM-11 Extended Commands

This chapter describes the DSM-11 extended commands in alphabetical order and provides examples of their use.

5.1 Introduction To DSM-11 Extended Commands

A DSM-11 command is a name for the action the command performs. DSM-11 has two types of commands:

- ANSI Standard commands
- Extended commands

ANSI Standard commands are specified in the *ANSI MUMPS Language Standard* and follow Standard usage. The Standard reserves the letters A through Y as the first letters of Standard command names.

Extended commands, as specified in the *ANSI MUMPS Language Standard*, are implementation-specific additions to the language. Extended commands always begin with the letter Z. This chapter lists only the extended commands.

You can abbreviate any extended command name to its first two letters (the Z and the second letter of the command name).

Many DSM-11 commands can take one or more arguments. Arguments are expressions or expression atoms (for example, a function and its arguments or a variable name) that define and control the action of the command.

Some DSM-11 commands never take arguments. Their action needs no further definition than a specification of their name.

Still other DSM-11 commands take arguments only in certain circumstances. Such commands change their meaning depending on whether or not you specify an argument or argument list.

5.2 Extended Command Descriptions

The following pages contain reference descriptions of all DSM-11 extended commands. Each command description contains an explanation of the purpose, forms, and operation of the command. The descriptions are in alphabetical order for ease of referencing.

The descriptions also include one or more examples of command usage. All command-line examples are presented as they would be on entry at a terminal. All routine-line examples are presented as they would be when listed on a line printer.

ZALLOCATE

PURPOSE:

ZALLOCATE makes specified variables or specified nodes of variables unavailable for allocation (locking) by other users.

FORM:

ZA{LLOCATE}{:*postcond*}SP*argument*{:*timeout*},...

In which *argument* can be one of the following:

name

@expr atom

where:

postcond	is a postconditional expression
name	is the name of a local or global variable with or without subscripting
@expr atom	is an indirect reference that evaluates to one or more ZALLOCATE arguments
timeout	is a timeout that specifies how many seconds DSM-11 is to try to lock the variable

EXPLANATION:

ZALLOCATE locks the variable(s) specified. If you use multiple arguments, ZALLOCATE locks all variables specified. When you lock a variable using ZALLOCATE, DSM-11 enters its name into a table that prevents any other user from entering a LOCK or ZALLOCATE command for that variable. It does not unlock any variable you previously locked with either LOCK or ZALLOCATE.

Consider the following statements:

```
ZA ^A, ^B,^C


ZA ^D, ^E
```

ZALLOCATE (Cont.)

DSM-11 interprets these statements as:

- Allocate (lock) ^A
- Allocate (lock) ^B
- Allocate (lock) ^C
- Allocate (lock) ^D
- Allocate (lock) ^E

DSM-11 locks all five global variables. Unlike the LOCK command, ZALLOCATE does not implicitly unlock any previously locked variables.

If you lock a local variable, ZALLOCATE locks all local variables of that name on the system. If you lock a global variable, ZALLOCATE locks that global variable for your UCI.

If you lock an unsubscripted variable, DSM-11 prevents any other user from locking a subscripted variable with the same name. If you lock a subscripted variable, DSM-11 prevents any other user from locking any descendants or direct ancestors of the node you specify.

You must always use full references or indirection to full references as arguments for the ZALLOCATE command. DSM-11 does not allow naked references with ZALLOCATE.

DSM-11 unlocks a variable locked with ZALLOCATE when any of the following actions occur:

- You enter a ZDEALLOCATE command with that variable specified as an argument.
- You enter an argumentless ZDEALLOCATE command.
- You enter a HALT command.

COMMENTS:

Keep the following points in mind when you use the ZALLOCATE command:

1. Like LOCK, ZALLOCATE is a convention. It does not prevent multiple simultaneous accesses of a variable. Any user who does not follow the convention of issuing ZALLOCATE commands and avoiding work on locked variables can modify variables for which you have issued a ZALLOCATE command.

ZALLOCATE (Cont.)

2. If you attempt to lock a variable or variables already locked by another user, DSM-11 suspends execution of your routine until the variable or variables are unlocked. To overcome such a potentially long suspension of execution, use a timeout with ZALLOCATE.

   If DSM-11 cannot lock the variable before the timeout occurs, it sets $TEST to false (zero) and resumes execution. If DSM-11 can lock the variable during the timeout, it locks the variable, sets $TEST to true (one), and resumes execution. DSM-11 resumes execution as soon as it can lock the variable. It ignores any time remaining on the timeout.

3. You cannot use one locking command to lock variables already locked by the other. A ZALLOCATE command cannot lock variables previously locked by another user with LOCK. A LOCK command cannot lock variables previously locked by another user with ZALLOCATE.

   Consider the following situation. User A issues a LOCK command to lock ^X:

   ```
   L ^X
   ```

   User B then issues a ZALLOCATE for the same global:

   ```
   ZA ^X
   ```

   In this case, DSM-11 suspends execution of user B and tries to lock ^X until user A unlocks ^X. If user B used a timeout, DSM-11 suspends execution until the timeout occurs.

4. You cannot use one locking command to unlock variables locked by the other. A ZALLOCATE command does not unlock any variables previously locked with LOCK. A LOCK command does not unlock any variables previously locked with ZALLOCATE.

   Consider the following statements:

   ```
   ZA ^A,^B,^C
   L ^D
   ```

   These statements lock the globals ^A, ^B, ^C, and ^D. The LOCK statement also unlocks any variables locked in a previous LOCK statement.

   Consider also these statements:

   ```
   L ^A
   ZA ^B,^C,^D
   L
   ```

ZALLOCATE (Cont.)

These statements lock the global ^A, and then lock the globals ^B, ^C, and ^D. The LOCK command with no argument then unlocks ^A, but does not affect ^B, ^C, or ^D, which remain locked.

RELATED:

The LOCK Command
The ZDEALLOCATE Command
The $TEST Special Variable

EXAMPLES:

The following example locks the global ^A with the LOCK command and locks the globals ^B, ^C, and ^D with the ZALLOCATE command. Because ZALLOCATE does not affect the status of any other locked references, the globals ^A, ^B, ^C, and ^D are all locked at the end of this sequence of commands.

```
L ^A


ZA ^B,^C,^D
```

The following example locks the globals ^B, ^C, and ^D with the ZALLOCATE command and locks the global ^A with the LOCK command. Because the LOCK command does not unlock any variables previously locked with ZALLOCATE, all specified globals are locked at the end of this series of commands.

```
ZA ^B,^C,^D


L ^A
```

ZBREAK

PURPOSE

ZBREAK turns on, turns off, and generally controls the DSM-11 debugger.

FORMS:

ZB{REAK}{:*postcond*}␣*control element*

where:

postcond is a postconditional expression

control element is ON, OFF, OVER, IN, or OUT

EXPLANATION:

The ZBREAK command controls the debugger and is used in conjunction with the BREAK command, the ZGO command, and the $ZBREAK special variable. The ZBREAK command is used in two ways:

- ZBREAK turns the debugger on or off, using the *control element* ON or OFF. The default value is for the debugger to be OFF. When ZBREAK is ON, the debugger recognizes the BREAK command and breakpoints set with the $ZBREAK special variable.

- ZBREAK can be used to control execution during a breakpoint. The ZBREAK command is used to continue execution in single-step mode. Enter the ZBREAK command with a *control element* of OVER, IN, or OUT.

  ZBREAK OVER treats any DO or XECUTE and any subroutine associated with the DO or XECUTE command as one unit. Execution is not halted at any of the commands in the subroutine. Execution is halted when another command is reached in the "top" routine.

  ZBREAK IN steps to the next command and issues a BREAK message.

  ZBREAK OUT steps to the first command following the QUIT from the current routine, subroutine, or XECUTE string.

COMMENTS:

Keep the following points in mind when using ZBREAK:

1. The *control element* is not a string literal. It must not have quotation marks (") around it.

ZBREAK (Cont.)

2. The *control element* is not an expression. You cannot set a variable equal to one of the strings OVER, IN, or OUT and then use that expression as an argument for ZBREAK.

3. You usually use ZBREAK OUT when you have stepped into a routine with ZBREAK IN, executed a number of steps, and then decided that you do not need to monitor the remainder of the subroutine.

4. You can use a carriage return in place of ZBREAK IN to advance to the next command. This has exactly the same effect as ZBREAK IN.

5. A breakpoint can also be generated with the CTRL/B key. See the *DSM-11 User's Guide* for more discussion of this key.

RELATED:

The BREAK Command
The ZGO Command
$ZBREAK Special Variable
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following use of ZBREAK turns on the debugger:

```
)ZBREAK ON
D)
```

The prompt (D>) indicates that the debugger is on and that all breakpoints will be honored. For more information on the debugger, see the *DSM-11 User's Guide*.

ZDEALLOCATE

PURPOSE:

ZDEALLOCATE unlocks variables locked with the ZALLOCATE command.

FORMS:

ZD{EALLOCATE}{:*postcond*}

ZD{EALLOCATE}{:*postcond*} SP *argument*,...

In which argument can be one of the following:

name

@expr atom

where:

postcond	is a postconditional expression
name	is the name of a local or global variable with or without subscripting
@expr atom	is an indirect reference that evaluates to one or more ZDEALLOCATE arguments

EXPLANATION:

The ZDEALLOCATE command without arguments unlocks all variables previously locked with ZALLOCATE. The ZDEALLOCATE command with arguments unlocks the variable(s) specified by the argument(s). ZDEALLOCATE with arguments does not unlock any variable not specified.

Neither form of ZDEALLOCATE unlocks variables previously locked with LOCK. That is, you can use ZDEALLOCATE only to unlock variables locked with ZALLOCATE.

RELATED:

The LOCK Command
The ZALLOCATE Command

ZDEALLOCATE (Cont.)

EXAMPLES:

The following example locks the globals ^A, ^B, and ^C with the LOCK command and locks the globals ^D and ^E with the ZALLOCATE command. It then issues an argumentless ZDEALLOCATE that unlocks the globals ^D and ^E. The globals ^A ^B, and ^C remain locked.

```
L ^A,^B,^C
ZA ^D,^E


ZD
```

The following example locks the globals ^A, ^B, and ^C with the ZALLOCATE command and then unlocks the global ^C with ZDEALLOCATE. The globals ^A and ^B remain locked.

```
ZA ^A,^B,^C


ZD ^C
```

ZGO

PURPOSE:

ZGO resumes execution of a routine after the execution of a BREAK command, or starts a debugging session at a specified entry reference.

FORM:

ZG{O}

ZG{O}{:*postcond*}SP{*entry ref*}

where:

postcond	is a postconditional expression
entry ref	is an entry reference specifying a line and/or routine

EXPLANATION:

ZGO is a debugging command. When DSM-11 encounters a breakpoint, it suspends execution and prints a message and a line reference showing where it executed the breakpoint. The breakpoint occurs when you are using the DSM-11 debugger, and can be caused by a BREAK command, a breakpoint set with the $ZBREAK special variable, or use of the CTRL/B key. See the *DSM-11 User's Guide* for more information on breakpoints and on the DSM-11 debugger.

After you enter a ZGO in a command line, DSM-11 continues execution with the statement after the breakpoint.

Using ZGO with a *entry ref* transfers control to that reference and causes a BREAK command to be issued before the first command is executed. This option allows you to start at the beginning of a routine (or at another point in the routine), and then to step through the routine using the ZBREAK command.

See the *DSM User's Guide* for more information on errors.

COMMENTS:

Keep the following points in mind when you use the ZGO command:

1. If you issue an argumentless ZGO from any context other than Break Mode (after a breakpoint), a <LVLER> error will result.

ZGO (Cont.)

2. If you issue an argumentless ZGO from a subroutine within Break Mode, the subroutine return is discarded, and execution resumes at the command following the breakpoint.

3. ZGO with an entry reference can be issued only from Programmer Mode but not from the Break Mode. If you issue a ZGO with an entry reference from Break Mode, a <LVLER> error will result.

RELATED:

The BREAK Command
The ZBREAK Command

EXAMPLES:

The prompt (DB<) indicates that the debugger is on and that you are within the debugger Break Mode.

The following example uses ZGO to resume execution after a BREAK.

```
)D ^TEST
<BREAK> A+2^TEST B:X=15
DB)ZG
```

The following example shows how ZG is used to step through a routine from the beginning:

```
)ZG ^STU
<BREAK)STU+1:1 I '$V($V(44)+35) W "Not in correct mode",! Q
DB)
```

ZINSERT

PURPOSE:

ZINSERT inserts a line into the routine currently in memory.

FORM:

ZI{NSERT}{:*postcond*}SP*argument*,...

In which *argument* can be one of the following:

string{:*line ref*}

string{:*line spec*}

@expr atom

where:

postcond	is a postconditional expression
string	is a string containing one or more DSM-11 statements
line ref	is a line reference to the line immediately preceding the position where you want to insert the new line
line spec	line is a line specification of the line immediately preceding the position where you want to insert the new line
@expr atom	is an indirect reference that evaluates to one or more ZINSERT arguments

EXPLANATION:

ZINSERT is an editing command. You can use it to insert a new line into the routine currently in your partition.

The argument for ZINSERT contains two parts. The first part is an expression that evaluates to a string containing the DSM-11 statements to insert. You must format the string as follows:

1. Do not use the TAB character as you would when entering a routine line. Use a single space character as either the first character of the string (if you do not include a line label) or as the single character separating the label from further text on the line (if you do include a line label).

ZINSERT (Cont.)

2. Do not insert a carriage return as the last character in the string.

   For example, to have DSM-11 store:

   TAB S SP A = 5

   enter the ZINSERT statement:

   ZI SP " SP S SP A = 5":*line*

   (*line* refers to a *line ref* or *line spec*.) To have DSM-11 store:

   RESET TAB S SP A = 5

   enter the ZINSERT statement:

   ZI SP "RESET SP S SP A = 5":*line*

3. Represent all quotes on the line to be inserted with two sets of quotation marks. For example, to have DSM-11 store:

   TAB W SP !!,"BEGIN TEST"

   enter the ZINSERT statement:

   ZI SP " SP W SP !!,""BEGIN TEST""":*line*

The string expression you use in the ZINSERT argument thus has the same form as a routine line returned by the $TEXT function.

The second part of the ZINSERT argument is a specification of the line immediately preceding the position where you want to insert the new line. The specification can be in the form of a line reference or a line specification.

The line reference must evaluate to the line label or the label and offset of the line. The line specification must evaluate to an integer value (preceded by a plus sign) that specifies the sequential position of the line in the routine.

After you enter the ZINSERT statement, DSM-11 moves an implicit line pointer to the end of the line specified. It then inserts the new line immediately after the line pointer. If you did not specify a line for the second part of the ZINSERT argument, DSM-11 inserts the string at the current position of the line pointer.

For example, to insert the line:

B TAB statement SP statement

ZINSERT (Cont.)

following line A in the routine:

A⟨TAB⟩;comment
C⟨TAB⟩statement⟨SP⟩statement

enter the ZINSERT statement:

ZI⟨SP⟩"B⟨SP⟩statement⟨SP⟩statement":A

The routine now contains:

A⟨TAB⟩;comment
B⟨TAB⟩statement⟨SP⟩statement
C⟨TAB⟩statement⟨SP⟩statement

COMMENTS:

Keep the following points in mind when using the ZINSERT command:

1. You should use ZINSERT only in command lines or in XECUTE arguments.

2. You can use ZINSERT in conjunction with the ZREMOVE command to replace an existing line. For example, if you want to replace line C in the routine:

   A⟨TAB⟩;comment
   B⟨TAB⟩statement⟨SP⟩statement
   C⟨TAB⟩statement⟨SP⟩statement

   with the line:

   C⟨TAB⟩statement

   enter a ZINSERT command to place the new line C just after the existing line C.

   ZI⟨SP⟩"C⟨SP⟩statement":C

   ZINSERT inserts the line after the line referenced. (DSM-11 accepts the line label C even though the line label is a duplicate.)

   The routine now has two lines labeled C.

   C⟨TAB⟩statement⟨SP⟩statement
   C⟨TAB⟩statement

Now, enter a ZREMOVE command to remove line C:

```
ZR SP C
```

ZREMOVE begins its search from the top of the routine. ZREMOVE encounters the first line C and removes it. The routine now contains only the correct line C.

3. To insert a line at the beginning of a routine, use a line specification of +0. For example:

```
ZI "INV ; INVENTORY UPDATE":+0
```

4. You can use both name and argument indirection in ZINSERT arguments.

5. The commands ZREMOVE and ZPRINT also move the implicit line pointer.

RELATED:

The XECUTE Command
The ZPRINT Command
The ZREMOVE Command
The $TEXT Special Variable
The Indirection Operation (Section 2.6.6)
Line References (Section 1.6.1)
Line Specifications (Section 1.6.3)
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following example uses $TEXT and ZINSERT to merge two routines, ROU1 and ROU2.

1. Load ROU2:

```
ZL ROU2
```

2. Place ROU2 into a global array:

```
F L=1:1 S T=$T(+L) Q:T=""  S ^TXT(L)=T
```

3. Remove ROU2 and load ROU1:

```
ZR
ZL ROU1
```

ZINSERT (Cont.)

4. Print ROU1 to determine the last line in the routine. (For the purposes of this example, assume the last line has the label END.)

5. Append the text from ROU2 to ROU1:

```
F I=1:1:L-1 ZI ^TXT(I)
```

6. Save the merged routine with ZSAVE.

ZLOAD

PURPOSE:

ZLOAD writes a routine from your routine directory or from the current device into memory.

FORMS:

ZL{OAD}{:*postcond*}

ZL{OAD}{:*postcond*}(SP)*argument*,...

In which *argument* can be one of the following:

routine

@expr atom

where:

postcond	is a postconditional expression
routine	is the name of the routine
@expr atom	is an indirect reference that evaluates to one or more ZLOAD arguments

EXPLANATION:

ZLOAD without arguments performs an opposite action from that of an argumentless ZPRINT command. It writes a routine from a device to memory.

To load a routine from a device, you must:

1. Execute an OPEN command to open the device
2. Execute a USE command to use the device
3. Execute one or more ZLOAD commands without arguments to load the routine from the device into memory

See the *DSM User's Guide* for more information on using devices.

ZLOAD (Cont.)

ZLOAD with an argument performs an opposite action from that of a ZSAVE command. It writes the specified routine from your routine directory into your partition.

In both cases, DSM-11 leaves the implicit line pointer at the end of the last line in the routine. (See the description of the ZINSERT command for more information on the implicit line pointer.)

COMMENTS:

Keep the following points in mind when using the ZLOAD command:

1. If you use a postconditional expression with ZLOAD, DSM-11 executes the ZLOAD only if the postconditional expression has a value of true.

2. You can use name or argument indirection with the ZLOAD argument. (In the case of ZLOAD, name and argument indirection are one and the same.)

3. When ZLOAD loads a routine, it deletes any routine currently in your partition.

4. You should use ZLOAD only in command lines or in XECUTE arguments. You should not use ZLOAD in routine lines.

RELATED:

The OPEN Command
The USE Command
The ZPRINT Command
The ZSAVE Command
Entry References (Section 1.6.2)
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following example loads the routine ROUT from the routine directory.

```
ZL ROUT
```

The following example loads the first routine from the device specified by the variable DEV.

```
O DEV U DEV ZL
```

ZPRINT

PURPOSE:

ZPRINT writes the current routine or routine lines on the current output device.

FORMS:

ZP{RINT}{:*postcond*}

ZP{RINT}{:*postcond*}SP*argument*,...

In which *argument* can be one of the following:

start ref{:*end ref*}

start spec{:*end spec*}

@expr atom

where:

postcond	is a postconditional expression
start ref	is a line reference that specifies the first line to write
end ref	is a line reference that specifies the last line to write
start spec	is a line specification of the first line to write
end spec	is a line specification of the last line to write
@expr atom	is an indirect reference that evaluates to one or more ZPRINT arguments

EXPLANATION:

ZPRINT without arguments is the opposite of the argumentless ZLOAD command. It writes the entire routine in your partition to the current output device. DSM-11 leaves its implicit line pointer at the end of the last line in the routine.

ZPRINT with a single line reference or line specification as an argument writes the line specified on the current output device. DSM-11 leaves the implicit line pointer at the end of the line printed and before the beginning of the first line not written.

ZPRINT (Cont.)

ZPRINT with two line references or line specifications separated by a colon as an argument writes all lines from *start ref* or *start spec* through *end ref* or *end spec* on the current output device. DSM-11 leaves its implicit line pointer after the end of the last line written.

If *end ref* or *end spec* does not exist in the current routine, DSM-11 writes all lines from *start ref* (or *start spec*) to the end of the routine.

COMMENTS:

If you use a postconditional expression with ZPRINT, DSM-11 executes the ZPRINT only if the postconditional expression has a value of true.

RELATED:

The ZINSERT Command
The ZLOAD Command
The ZREMOVE Command
The Indirection Operator (Section 2.6.6)
Line References (Section 1.6.1)
Line Specifications (Section 1.6.3)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example writes the routine in memory on the current output device.

```
ZP
```

The following example writes the first line of the routine in memory on the current output device.

```
ZP +1
```

The following example writes the section of the current routine beginning with the line labeled A1 and ending with the line labeled B1.

```
ZP A1:B1
```

The following example writes the first through tenth lines of the current routine.

```
ZP +1:+10
```

ZPRINT (Cont.)

The following example writes the line at offset +2 from the line labeled C.

```
S A="B",B="C"
ZP @A+2
```

The following example writes the second and fourth sequential lines. Then, it writes all lines from the sixth sequential line to the line labeled END (in theory, the last line in the routine). The routine does not need a line labeled END for the command to write the last routine line. If the label specified does not exist, DSM-11 assumes it to be a reference to text just past the last routine line.

```
ZP +2,+4,+6:END
```

The following example copies the routine ROUT from the routine directory to magnetic tape.

```
ZL ROUT O 47 U 47 W *5,"ROUT",! ZP    W !
```

ZQUIT

PURPOSE:

ZQUIT directs control to the previously declared error-handling routine.

FORM:

ZQ{UIT}{:*postcond*}

where:

postcond is a postconditional expression

EXPLANATION:

ZQUIT is used to invoke a series of error-handling routines in a set of nested DO or XECUTE statements. ZQUIT directs control to the previous error-handling routine declared in a lower level DO or XECUTE statement.

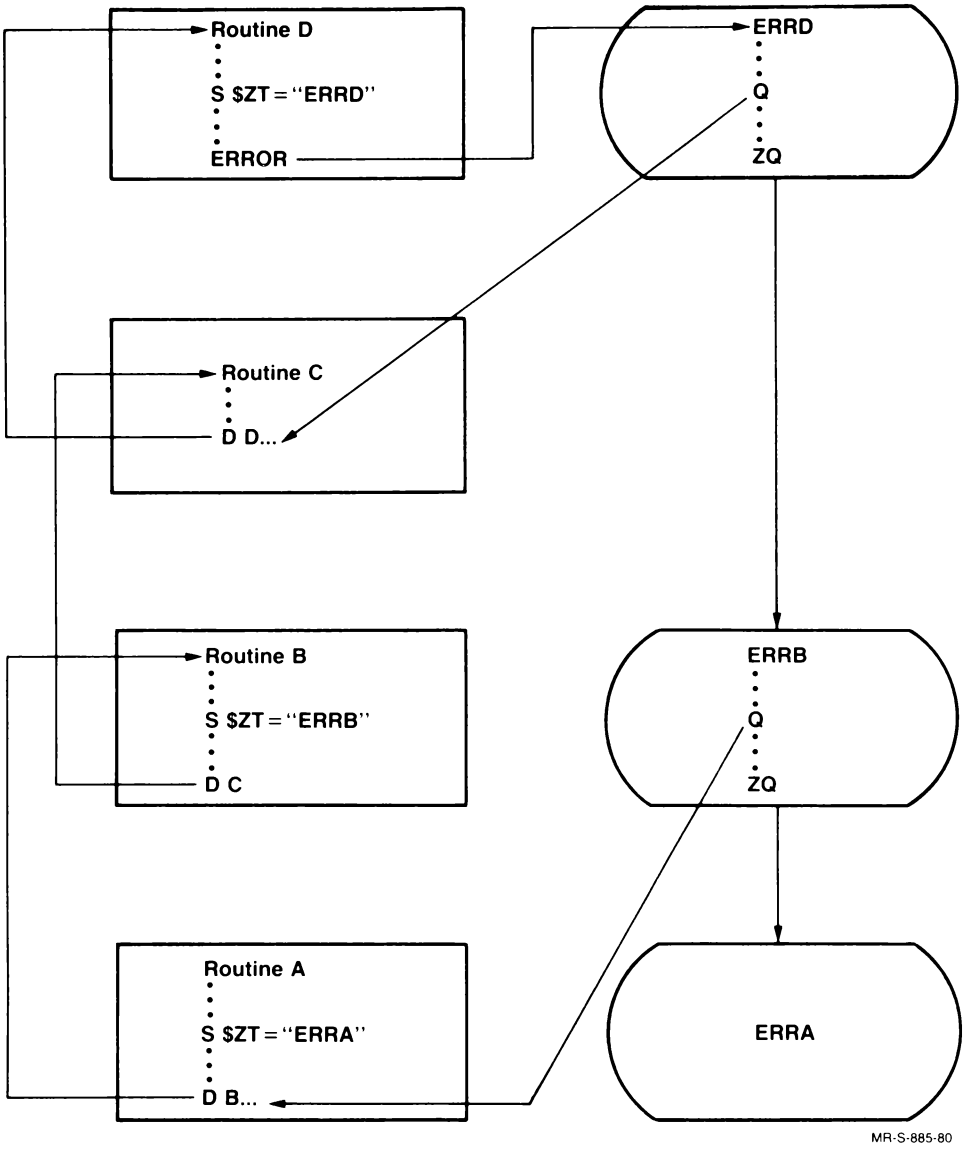
To understand how ZQUIT operates, consider a series of four routines called A, B, C, and D. Routine A contains a DO statement to call routine B; routine B contains a DO statement to call routine C; routine C contains a DO statement to call routine D.

Each of these routines can establish its own error-handling routine. In other words, each can set the $ZTRAP special variable to the name of a different error-handling routine to which DSM-11 passes control in the event of an error.

For example, suppose routine A sets $ZTRAP to the value "ERRA"; routine B sets $ZTRAP to the value "ERRB"; routine C does not set $ZTRAP; and routine D sets $ZTRAP to the value "ERRD".

By using the ZQUIT command in these error-handling routines, you can link all the error handlers for the nested series of A, B, C, and D. Figure 5-1 shows the flow of control in the event of an error.

Figure 5-1: Flow of Control with ZQUIT



As Figure 5-1 shows, if DSM-11 detects an error in routine D, it transfers control to ERRD, the error-handling routine designated by routine D in $ZTRAP.

ZQUIT (Cont.)

If ERRD can correct the error, it can exit normally with a QUIT statement or transfer control with a GOTO statement. QUIT returns control to the point following the DO argument that invoked the routine that caused the error. (For ERRD, that point is immediately following the DO argument in C that invoked routine D.) A GOTO statement redirects control to any specified point.

If ERRD cannot correct the error, you can use a ZQUIT in ERRD to direct control to the previously declared error-handling routine, ERRB for routine B. The ZQUIT will remove the context of routines D and C from the call stack. If ERRB cannot correct the error, a ZQUIT in ERRB directs control to the previously declared error-handling routine ---ERRA for routine A --- and removes the context of routine B from the call stack.

Thus, when used together, $ZTRAP and ZQUIT provide a mechanism that allows you to link all the error handlers in an application to form a network that cycles an error through each error handler until the error condition is resolved.

See the *DSM-11 User's Guide* for a complete description of error handling with ZQUIT and other error-processing options.

COMMENTS:

ZQUIT can be used at any time, even when no error has occurred.

RELATED:

The DO Command
The GOTO Command
The QUIT Command
The ZTRAP Command
The $ZERROR Special Variable
The $ZTRAP Special Variable
Postconditional Expressions (Section 2.7.4)

EXAMPLE:

The following example is a portion of an error handler that writes specific error messages. If the error handler encounters a NAKED error, it executes a ZQUIT command that transfers control to the error handler for the previous routine in the series. In all other cases, it prints a message, clears $ZERROR, and returns control to the point immediately following the DO argument that invoked the routine that caused the error.

```
L    ZQ:$P($ZE,")",1)["NAKED"
     U 0 W !,"SORRY BUT LINE", $P($P($ZERROR,")",2),"^",1)
     W " IN ROUTINE ",$P($P($ZERROR,"^",2),":",1)
     W " GENERATED AN ",$E($P($ZERROR,")",1)2,99)," ERROR"
```

ZREMOVE

PURPOSE:

ZREMOVE deletes the current routine or specified lines in the current routine.

FORMS:

ZR{EMOVE}{:*postcond*}

ZR{EMOVE}{:*postcond*}␣*argument*,...

In which *argument* can be one of the following:

start ref{:*end ref*}

start spec{:*end spec*}

@expr atom

where:

postcond	is a postconditional expression
start ref	is a line reference that specifies the first line to delete
end ref	is a line reference that specifies the last line to delete
start spec	is a line specification of the first line to delete
end spec	is a line specification of the last line to delete
@expr atom	is an indirect reference that evaluates to one or more ZREMOVE arguments

EXPLANATION:

ZREMOVE without arguments deletes the entire routine from your partition. It does not affect the value of any local variables the routine defined. That is, those local variables are still listed in your local symbol table.

ZREMOVE with arguments deletes only the specified lines from the routine. ZREMOVE with one line reference or line specification as an argument deletes the routine line specified. ZREMOVE with two line references or line specifications separated by a colon deletes all lines from the first specified through the second specified.

ZREMOVE (Cont.)

In either case, DSM-11 sets its implicit line pointer at the line before the first line you deleted in the last (or only) ZREMOVE argument. You can now enter new lines just below the last line left undeleted.

If the first line reference or specification follows the second line reference or specification in the routine, DSM-11 does not delete any lines. If the second line reference or specification is not defined (does not exist), ZREMOVE deletes all lines from the first line reference or specification to the end of the routine.

COMMENTS:

You should keep the following points in mind while using ZREMOVE:

1. You should use ZREMOVE only in command lines or in XECUTE arguments. You should not use ZREMOVE in routine lines.

2. You create certain side effects if you use multiple ZREMOVE arguments to remove a series of lines. For example, if you want to delete lines A+1 and A+2 in a routine, you should not use the statement:

   ```
   ZR A+1,A+2
   ```

   This statement removes the lines A+1 and A+3. That is, the first argument removes line A+1. Line A+2 now becomes line A+1. The second argument to remove line A+2 thus actually removes the old line A+3.

   To remove lines A+1 and A+2, issue the statement:

   ```
   ZR A+1,A+1
   ```

   or the statement:

   ```
   ZR A+1:A+2
   ```

3. You can use ZREMOVE in conjunction with the ZSAVE command to delete a routine from your routine directory. To do so, take the following steps:

   a. To save the routine in your partition, execute a ZSAVE command.

   b. Execute a ZREMOVE command without arguments to clear your partition:

      ```
      ZR
      ```

ZREMOVE (Cont.)

c. Execute a ZSAVE command with the name of the routine you want to delete as an argument. For example, if you want to delete the routine TEST, type:

```
ZS TEST
```

DSM-11 deletes the routine and removes its name from your routine directory.

RELATED:

The ZINSERT Command
The ZPRINT Command
The ZSAVE Command
The $TEXT Function
The Indirection Operator (Section 2.6.6)
Line References (Section 1.6.1)
Line Specifications (Section 1.6.3)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example deletes the routine currently in the partition.

```
ZR
```

The following example deletes the third line below the line labeled ADDR.

```
ZR ADDR+3
```

The following example deletes all lines from that labeled A1 through that labeled B1.

```
ZR A1:B1
```

ZSAVE

PURPOSE:

ZSAVE writes the routine currently in memory into your routine directory.

FORMS:

ZS{AVE}{:*postcond*}

ZS{AVE}{:*postcond*}␣*argument*,...

In which *argument* can be one of the following:

routine

@expr atom

where:

postcond	is a postconditional expression
routine	is the name of the routine to save
@expr atom	is an indirect reference that evaluates to or more ZSAVE arguments

EXPLANATION:

ZSAVE writes the routine in your partition into your routine directory. If you have previously associated a name with the routine, you can enter ZSAVE without an argument.

If you have not previously associated a name with the routine, or if you want to save the routine under a new name, you must use a name under which you want to store the routine as a ZSAVE argument. If you do not enter a name for an unnamed routine, DSM-11 returns an error.

Examine $TEXT(+0) to determine what name (if any) is associated with the routine in memory.

See the *DSM User's Guide* for more information on errors.

COMMENTS:

Keep the following points in mind when you use the ZSAVE command:

1. ZSAVE does not affect the implicit DSM-11 line pointer.

2. You can use ZSAVE in conjunction with the ZREMOVE command to delete a routine from your routine directory. To do so, take the following steps:

   a. Execute a ZSAVE command if you want to save the routine currently in memory.

   b. Execute a ZREMOVE command without arguments to clear your work area:

```
ZR
```

   c. Execute a ZSAVE command with the name of the routine you want to delete as an argument. For example, if you want to delete the routine TEST, type:

```
ZS TEST
```

   DSM-11 deletes the routine and removes its name from your routine directory.

RELATED:

The ZLOAD Command
The ZREMOVE Command
The $TEXT Function
The Indirection Operator (Section 2.6.6)
References (Section 1.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example saves a routine in memory that already has a name associated with it.

```
ZS
```

The following example saves a routine in memory that does not have a name associated with it. (The following example could also be saving a previously named routine under a new name.)

```
ZS PROG
```

The following example uses indirection to save the current routine under the names AA and AAA.

```
S A="AA",B="AAA",X="@A,@B"
ZS @X
```

ZTRAP

PURPOSE:

ZTRAP forces an error when ZTRAP is encountered by the interpreter.

FORM:

ZT{RAP}(SP){*string*}

where:

string	is an informational message to be included in the error message string

EXPLANATION:

When DSM-11 encounters a ZTRAP command in a DSM routine, it forces a ZTRAP error. This error prevents the routine from executing any further.

The informational message can be any four characters, and is then included in the error message, preceded by the letter Z.

For example, if you issue the following ZTRAP statement:

```
ZTRAP ERR1
```

the resulting ZTRAP error message is:

<ZERR1>label+offset ^routine name:command number

If you issue ZTRAP command without an argument, then the result is:

<ZTRAP>label+offset^routine name:command number

COMMENTS:

The ZTRAP command provides a way to gracefully abort the execution of an application routine. DSM-11 reports the ZTRAP error in the $ZERROR special variable. Thus, you can test for this error in an error-processing routine, and take the appropriate steps to terminate your application when the error occurs.

The ZTRAP command produces a trap to a declared trap or error handler. The command is particularly useful in instances where it is desirable to redirect program flow to a lower level call frame.

ZTRAP (Cont.)

If no argument is present, DSM-11 substitutes TRAP. If an argument is present, DSM-11 use the first four characters only of the string given as an argument to the ZTRAP command.

RELATED:

The QUIT Command
The ZQUIT Command
The $ZERROR Special Variable
The $ZTRAP Special Variable

EXAMPLE:

The following example generates a ZTRAP error when a user types the "E" key. Control passes to an error-processing routine (set in the $ZTRAP special variable) that runs down the application.

```
STRT    S $ZT="ERR1^EXIT"


        .
        R "Enter choice or 'E' to exit: ",CH
        I CH="E" ZT


        .
ERR1    I $ZE["ZTRAP" H
```

ZUSE

PURPOSE:

ZUSE allows you to write to a terminal-type device owned by another user.

RESTRICTION:

The ZUSE command may be restricted on your system. If so, you can only use ZUSE if you are in the system account or if you are running a library routine that contains ZUSE statements.

See your system manager to determine if ZUSE is restricted. See the *DSM User's Guide* for more information on library routines.

FORM:

ZU{SE}{:*postcond*} SP *argument*

In which *argument* can be one of the following:

terminal

@expr atom

where:

postcond	is a postconditional expression
terminal	is a device specifier of the terminal-type device
@expr atom	is an indirect reference that evaluates ZUSE argument

EXPLANATION:

ZUSE allows you to set any terminal-type device as the current I/O device without having to execute OPEN and USE commands. The terminal-type devices you can use include the line printer but exclude the DMC11.

After you execute a ZUSE, you can execute WRITE commands to that device:

```
ZU 10 W !,"LINE PRINTER OUT OF PAPER"
```

WRITE commands are the only commands you can give when you use ZUSE. The system arbitrates which user accesses the device at any one time.

You can use your principal device again with:

```
U 0
```

ZUSE (Cont.)

COMMENTS:

ZUSE is primarily intended for use in a broadcast utility routine for sending messages to other system users.

During SYSGEN you may disable ZUSE on a terminal by terminal basis. If this is done any message sent by ZUSE to a disabled terminal is ignored. You may want to do this for a terminal that is a printer and is not being used interactively by a user.

See the *DSM User's Guide* for the valid device specifiers you can use as ZUSE arguments.

RELATED:

The CLOSE Command
The OPEN Command
The USE Command
The WRITE Command
The Indirection Operator (Section 2.6.6)
Postconditional Expressions (Section 2.7.4)

EXAMPLES:

The following example writes a message on the terminal specified earlier in DEV.

```
ZU DEV W !,"ERRMON: LP ERROR DETECTED"
```

The following example prompts for a device specification and a message. Then it sends the message to the specified device.

```
DEV     R !,"ENTER DEVICE: ",DEVA
        R !,"ENTER MESSAGE: ",MSG
        ZU DEVA W !,MSG
```

ZWRITE

PURPOSE:

ZWRITE writes the contents of your local symbol table to the current device.

FORM:

ZW{RITE}{:*postcond*}

ZW{RITE}{:*postcond*}(SP)*argument*}

In which *argument* can be one of the following:

local

@expr atom

where:

postcond	is a postconditional expression
local	is the name of a local variable with or without subscripting
@expr atom	is an indirect reference that evaluates to one or more ZWRITE arguments

EXPLANATION:

ZWRITE without an argument writes all currently defined local variables in numeric collating order. When ZWRITE has a variable name as an argument, it writes only that variable. If the variable has defined nodes, then it writes all the nodes. When ZWRITE has a variable name and subscript as an argument, it writes only the specified node and its descendant nodes.

RELATED:

Array Structure (Section 2.4.4)
Local Variables (Section 2.4.1)
Postconditional Expressions (Section 2.7.4)

ZWRITE (Cont.)

EXAMPLE:

The following example displays the contents of the symbol table.

```
ZW

A="JOHN JONES"
A(1)="444 EAST ELM ST"
B="123047702"
C="8779494"
```

The following example writes the contents of the local variables B and C:

```
S VAR="B,C"
ZW @VAR

B="123044702"
C="8779494"
```

The following example writes all defined elements of A:

```
ZW A

A(1)="1"
A(1,1)="2"
A(1,3)="3"
A(3)="6"
A(3,2)="2"
A(3,4,5)="5"
```

The following example writes all defined elements of A containing 3 as the first subscript:

```
ZW A(3)

A(3)="6"
A(3,2)="2"
A(3,4,5)="5"
```

Chapter 6
DSM-11 Functions

This chapter describes the DSM-11 functions and provides examples of their use.

6.1 Introduction To DSM-11 Functions

Functions perform a specified operation and return a value resulting from that operation. Each function consists of a mnemonic name that describes the function performed preceded by a dollar sign ($) and followed by one or more arguments.

The arguments specify the values that DSM-11 must use when it evaluates the function. Function arguments are enclosed in parentheses and immediately follow the function name. Multiple function arguments are separated from each other by commas and no intervening spaces. Except for the DSM-11 "trace" functions ($DATA, $NEXT, $ORDER, $ZNEXT, $ZORDER, $ZSORT), function arguments can always be expressions.

DSM-11 has two types of functions:

- ANSI Standard functions
- Extended functions

The ANSI Standard functions are specified by the *ANSI MUMPS Language Standard* and follow standard usage. The *ANSI MUMPS Language Standard* reserves the characters $A through $Y as the first characters of the standard function names.

As described in the *ANSI MUMPS Language Standard*, extended functions are implementation-specific extensions to the language. Extended function names start with the characters $Z.

You can abbreviate ANSI Standard function names to their first two characters (where the "$" counts as the first of those characters). You can abbreviate extended function names to their first three characters.

6.2 Function Descriptions

The following pages contain descriptions of all DSM-11 functions. Each function description contains an explanation of the purpose, forms, and operation of the function. The descriptions are in alphabetical order for ease of referencing.

The descriptions also contain one or more examples of how to use the function. All command-line examples are presented as they would appear when entered from a terminal. All routine-line examples appear as they would when listed on a line printer.

$ASCII

PURPOSE:

$ASCII returns the decimal ASCII code for the specified character in the specified string.

FORM:

$A{SCII}(*string*{,*position*})

where:

string	is a string
position	is an integer-valued expression that specifies the position in the string of the character whose value $ASCII is to return

EXPLANATION:

Each ASCII character in a string has a unique position number. In a string n characters long, the first, leftmost character has position 1; the last, rightmost character has position n.

$ASCII returns the decimal ASCII value of the character at the specified position in the specified string. $ASCII can return the decimal value of any character in the ASCII character set (that is, $A returns values between 0 and 255).

If the specified string is empty or if the integer value in the position argument is larger than the number of characters in the string, $ASCII returns -1. If you do not include the position argument, $ASCII returns the ASCII decimal value of the first character in the string.

COMMENTS:

You can use noninteger numeric values in the position argument. However, DSM-11 ignores the fractional portion and considers only the integer portion of the argument.

```
S Z="ABCDE" W $A(Z,3.5)
67
```

RELATED:

The $CHAR Function
The READ Command (READ *)
The WRITE Command (WRITE *)
The ASCII Character Set (Appendix A)

$ASCII (Cont.)

EXAMPLES:

The following example determines the ASCII decimal value of the character W.

```
W $A("W")
87
```

The following example determines the decimal equivalent of the ASCII value for the third character in the variable Z.

```
S Z="TEST" W $A(Z,3)
83
```

The following example returns a -1 because the second argument specifies a position greater than the number of characters in the string.

```
S Z="TEST" W $A(Z,5)
-1
```

The following example generates a simple checksum for the string X. When $CHAR(CS) is concatenated to the string, the checksum of the new string is always zero. Thus, validation is simplified.

```
CXSUM   S CS=0 F I=1:1:$L(X) S CS=CS+$A(X,I)
        S CS=128-CS#128
```

The following example converts a lowercase or mixed case alphabetic string to all upper case. The example uses three functions described in detail later in this chapter.

```
ST      S LEN=$L(STRING),NSTRING="" F I=1:1:LEN D CNVT
        Q
CNVT    S CHAR=$E(STRING,I) S:$A(CHAR))96 CHAR=$C($A(CHAR)-32)
        S NSTRING=NSTRING_CHAR
        Q
```

$CHAR

PURPOSE:

$CHAR returns the character(s) whose decimal ASCII value(s) you specify in the argument(s).

FORM:

$C{HAR}(*ASCII code*{,...})

where:

ASCII code is an integer expression that evaluates to the ASCII code for the character to be returned

EXPLANATION:

The arguments you can use are integer expressions whose values are in the range of 0 through 255 (the range of decimal codes for the entire ASCII character set).

COMMENTS:

Keep the following points in mind when you use the $CHAR function:

1. If you enter a value less than zero as an argument, DSM-11 returns a null string.

2. If you enter an argument list whose translated value exceeds 255 characters, DSM-11 generates an error.

3. You can use noninteger numeric values as arguments. However, DSM-11 ignores the fractional portion of the argument and considers only the integer portion as an ASCII code, for example:

```
W $C(65.5)
A
```

   In this example, $CHAR ignores the fractional portion of the number and produces the character represented by ASCII code 65, an uppercase A.

$CHAR (Cont.)

4. You should not use $CHAR to perform formatted writes. The $CHAR statements:

```
W $C(12) ;form feed
W $C(13) ;carriage-return/line-feed
```

   generally do not perform the same operation as the statements:

```
W #
W !
```

5. You can use $CHAR to generate escape sequences on video display terminals. (The third example shows how to do this on a VT100 terminal. See your user's guide for your terminal for details.)

RELATED:

The $ASCII Function
The READ Command (READ *)
The WRITE Command (WRITE*)
The ASCII Character Set (Appendix A)

EXAMPLES:

The following example generates the ASCII character represented by decimal code 72.

```
S X=72 W $C(X)
H
```

The following example tests $CHAR by comparing the returned characters (represented by ASCII decimal codes 32 to 95) with their equivalents in string A.

```
A   S A1=""
    F I=32:1:95 S A1=A1_$C(I)
    S A=" !""#$%&'()*+,-./0123456789:;<=>?
      ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_"
    I A1'=A W !,"ERROR A1=",A1
```

$CHAR (Cont.)

The following example generates escape sequences to control formatting on a VT100 video terminal.

```
STRT   S HOME=$C(27)_"[H"
       S ERSCRN=$C(27)_"[2J"
       S PRMT(5)="NAME: ",PRMT(7)="STREET: ",PRMT(9)="CITY: "
       S PRMT(11)="STATE: ",PRMT(13)="ZIP CODE: "
ESC    W HOME,ERSCRN
       F LIN=5:2:13 W $C(27)_"["_LIN_";30f",PRMT(LIN) R DATA(LIN)
       Q
```

To generate VT100 escape sequences using $CHAR, you specify ESC[the VT100 *Control Sequence Introducer* (CSI) as $C(27)_"[". The actual escape sequence codes follow the CSI. These codes generally consist of alphabetic characters that can be preceded by integers.

The sequence ESC[H specifies the "home" position on the VT100 terminal (the upper left corner of the display screen). The sequence ESC[2J erases all characters from the display screen. The sequence ESC[n1;n2f specifies direct cursor positioning when two integer-valued expressions precede the "f" and are separated from each other by a semicolon (;). The first integer (n1) specifies line position, and the second integer (n2) specifies column position. The first integer specifies the horizontal line number on the terminal screen; the second integer specifies the vertical column number.

Line ESC + 1 in the preceding example positions the cursor to line 5 column 30, writes a prompt, and reads the data entered in response to the prompt. Then, the FOR statement increments the line and prompt numbers by two; and the entire sequence repeats until each of the five prompts has been written and the associated data has been entered.

$DATA

PURPOSE:

$DATA returns an integer that indicates whether a specified node (element) contains data, has descendants (has a pointer to a node on the next lower level), both, or neither.

FORM:

$D{ATA}(*storage ref*)

where:

storage ref — is the name of a local or global variable or is an indirect reference that evaluates to the name of a variable or to another indirect reference

EXPLANATION:

In the integer value $DATA returns, the low-order digit is a truth value that tells if the variable has a value. The high-order digit is a truth value that tells if the variable is a node with a descendant.

The integer values $DATA can return are as follows:

Value	Meaning
0	The variable has neither a value nor descendants (that is, the variable does not exist).
1	The variable has a value but no descendants.
10	The variable has no value but has descendants.
11	The variable has a value and has descendants.

RELATED:

The KILL Command
The SET Command
The $NEXT Function
The $ZNEXT Function
The $ZSORT Function
Naked References (Section 2.4.5)

$DATA (Cont.)

EXAMPLES:

The following example sets nodes in the global array Z. All are on the second (single-subscript) level. Then the example tests ^Z(1) and determines that it contains data and no downward pointer.

```
K ^Z
S ^Z(1)="A",^Z(3)="B",^Z(4)="C"
W $D(^Z(1))
1
```

Given the first statement in the previous example, the following example gives a third (two-subscript) level node a value, then tests the system-created second-level node that points to it. The test shows that the second-level pointer node contains no data but has descendants.

```
S ^Z(2,3)="D" W $D(^Z(2))
10
```

The following example uses indirection to determine the attributes of a node.

```
S Y="^X(1,1)"
W $D(@Y)
10
```

$EXTRACT

PURPOSE:

$EXTRACT returns a substring from a specified position in a string.

FORM:

$E{XTRACT}(*string*{,*start pos*{,*end pos*}})

where:

string	is the string from which you want to extract a substring
start pos	is an integer-valued expression that specifies the sequential position of the first (or only) character in the substring you want to extract
end pos	is an integer-valued expression that specifies the sequential position of the last character in the substring you want to extract

EXPLANATION:

$EXTRACT has a one-, two-, and three-argument form.

In the one-argument form, $EXTRACT returns the first character from the specified string.

In the two-argument form, $EXTRACT returns the character at the position specified in *start pos* from the specified string.

In the three-argument form, $EXTRACT returns all characters from the position specified in *start pos* to the position specified in *end pos*. (In a string n characters long, the first, leftmost character is position one. The rightmost character is position n.)

The values you specify for *start pos* and *end pos* should be integer-valued expressions. If you use decimal numbers, DSM-11 strips away the decimal portion and uses only the integer portion as the position specifier, for example:

```
S Y="ABCDE" W $E(Y,1.99)
A
```

$EXTRACT (Cont.)

COMMENTS:

Keep the following points in mind when you use the $EXTRACT function:

1. The one-argument form:

   The one-argument form is equivalent to the two-argument form when the value of *start pos* is one.

2. The two-argument form:

   If the integer value of *start pos* is less than one or greater than the number of characters in the string, $EXTRACT returns a null string.

3. The three-argument form:

   1. If the integer value of *start pos* is greater than the integer value in *end pos*, $EXTRACT returns a null string.

   2. If the integer values in *start pos* and *end pos* are equal, $EXTRACT returns the character at the position both arguments specify. (Thus, this is equivalent to the two-argument form.)

   3. If the value of *end pos* is greater than the number of characters (positions) in the string, $EXTRACT returns all characters from the position specified in *start pos* to the end of the string.

   4. If the value of *start pos* is less than one, $EXTRACT returns all characters from the first character to the position specified by *end pos*.

RELATED:

The $FIND Function
The $LENGTH Function
The $PIECE Function

EXAMPLES:

The following example returns the fourth character in the string.

```
S X="THIS IS A TEST" W $E(X,4)
S
```

$EXTRACT (Cont.)

The following example shows that the one-argument form is equivalent to the two-argument form when *start pos* is one.

```
S STR="HELLO"
W $E(STR,1),!,$E(STR)
H
H
```

The following example returns a substring composed of the first through sixth characters.

```
S X="THIS IS A TEST" W $E(X,1,6)
THIS I
```

The following example returns a substring. Because *start pos* is less than zero, $EXTRACT returns a substring composed of the first through seventh characters.

```
S X="THIS IS A TEST" W $E(X,-1,7)
THIS IS
```

The following example returns a null string because *start pos* is larger than *end pos*.

```
S X="THIS IS A TEST" W $E(X,6,2)
```

The following example displays the strings held by X(1) through X(5) vertically on the current I/O device.

```
S X(1)="THIS",X(2)="IS A TEST",X(3)="FOR $E"
S X(4)="",X(5)=""
F I=1:1:10 W ! F J=1:1:5 W ?5*J,$E(X(J),I)
```

$FIND

PURPOSE:

$FIND returns an integer specifying the end position of a specified substring within a specified string.

FORM:

$F{IND}(*string,substring*{,*position*})

where:

string	is the string that contains the substring
substring	is the substring whose end position you want to determine
postion	is an optional integer-valued expression that specifies the sequential character position in the string from which $FIND must begin its search

EXPLANATION:

Each character in a string has a unique position number. In a string n characters long, the first character has position number 1 and the last character has position number n.

$FIND searches *string* for *substring*. If you do not specify *position*, $FIND begins its search with the first character in *string*. If you do specify *position*, $FIND begins its search at the character in that position in *string*.

If $FIND does not find *substring*, it returns a zero. If $FIND does find *substring*, it returns an integer value that represents the position of the character immediately to the right of *substring*.

$FIND ends its search with the first, leftmost occurrence of *substring* and returns only the position of the character immediately to the right of that occurrence. To find the nth occurrence of *substring* within *string*, you can use *position* and repeated searches to exclude earlier occurrences of *substring*.

COMMENTS:

Keep the following points in mind when you use the $FIND function:

1. You can use integer values of one or greater in *position*. If you use decimal numbers, DSM-11 ignores the decimal portion of the number and considers only the integer portion.

2. If $FIND locates no occurrence of *substring*, it returns a value of zero.

$FIND (Cont.)

3. If you specify a null string as *substring*, the two-argument form of $FIND returns a one and the three-argument form returns the value of *position*. If its value of position is greater than the number of characters in the string, the three-argument form of $FIND returns a zero.

4. If *string* is a null string, both forms of $FIND return a zero.

RELATED:

The $EXTRACT Function
The $LENGTH Function
The $PIECE Function

EXAMPLES:

The following example returns the position of the character immediately to the right of the substring FOR.

```
S X="FOREST" W $F(X,"FOR")
4
```

The following example returns the position of the character immediately to the right of the substring contained in the variable Y.

```
S X="FOREST",Y="FOR" W $F(X,Y)
4
```

The following example returns the position of the character immediately following the first occurrence of R after the seventh character in X.

```
S X="EVERGREEN FOREST",Y="R" W $F(X,Y,7)
14
```

The following examples illustrate the behavior of the null string with $FIND. If you specify the null string in the second argument, $FIND returns the position of the first character in the two-argument form and the position specified by the third argument in the three-argument form. When the third argument specifies a position greater than its number of characters in the string, $FIND returns a zero.

```
S X="FOREST" W $F(X,"")
1
W $F(X,"",4)
4
W $F(X,"",9)
0
```

$FIND (Cont.)

The following example uses name indirection to return the position of the character immediately to the right of the substring THIS.

```
S Y="X",X="""THIS IS A TEST"""
W $F(@Y,"THIS")
6
```

$JUSTIFY

PURPOSE:

$JUSTIFY returns a specified string right-justified in a field of a specified length.

FORMS:

$J{USTIFY}(*string,field*{*,frac field*})

where:

string	is the string to be right-justified
field	is an integer-valued expression that specifies the total length of the field in which to right-justify the first argument
frac field	is an integer-valued expression that specifies the number of fractional digits (including possible trailing zeros)

EXPLANATION:

The two-argument form of $JUSTIFY returns *string*right-justified in a field of spaces whose total length you specify in *field*. Thus, if *string* is four characters long and *field* is nine, $JUSTIFY returns *string* with five spaces concatenated to the left of the leading character in *string*.

The three-argument form of $JUSTIFY converts *string* to a numeric expression and returns *string* as a decimal number in a field whose total length you specify in *field* and whose fractional length (that is, the number of spaces to the right of the decimal point) you specify in *frac field*.

DSM-11 does this by performing the following steps:

1. It converts the value of the string to a numeric.

2. It appends zeros after the decimal point (and puts in the decimal point if necessary) to satisfy the *frac field* requirements.

3. It appends one zero to the left of the decimal point if the numeric value of *string* is less than 1.0 and greater than -1.0 and *string* does not already have a zero in this position.

4. It returns the string resulting from steps 1, 2,and 3. If the resulting string is not long enough to satisfy *field*, DSM-11 appends spaces to the left of the string until the total length of *string* equals *field*.

$JUSTIFY (Cont.)

COMMENTS:

Keep the following points in mind when you use the $JUSTIFY function:

1. In the three-argument form, the length of *field* includes the decimal point and, possibly, the sign of the numeric value, and a leading zero. Thus, if you enter the statement:

   ```
   W $J(-1234,9,2)
   ```

   DSM-11 returns:

   ```
   -1234.00
   ```

   (The first character on the line is a space.)

2. In either the two- or three-argument form, $JUSTIFY returns *string* unjustified if you enter a *field* value less than the number of characters in *string*.

3. You can use *frac field* to round decimal fractions. If you set the *frac field* to zero, $JUSTIFY rounds the integer portion of *string*:

   ```
   W $J(11.9,2,0)
   12
   ```

   If you set *frac field* to a positive integer value less than the number of nonzero fractional digits in *string*, $JUSTIFY returns the decimal portion of *string* rounded up if the last, excluded digit is greater than or equal to five.

   ```
   W $J(15.55,5,1)
   15.6
   ```

4. A negative value for *frac field* is invalid.

5. The *field* and *frac field* specifications do not guarantee that the length of the returned string will be less than or equal to either the *field* or *frac field* value. For example:

   ```
   W $J(.5555,2,3)
   0.556
   ```

RELATED:

The $LENGTH Function

$JUSTIFY (Cont.)

EXAMPLES:

The following example right-justifies the value of X in a 10-character field.

```
S X=14.445 W $J(X,10)
    14.445
```

The following example does not right-justify the value of X because the field specified in the second argument is shorter than the length of X.

```
S X=14.445 W $J(X,4)
14.445
```

The following example justifies the value of X in a 10-character field. Because the value of the third argument specifies four decimal places, $JUSTIFY adds a trailing zero to the decimal portion of the value of X.

```
S X=14.445 W $J(X,10,4)
   14.4450
```

The following example justifies the value of X in a 10-character field. Because the value of the third argument specifies two decimal places, $JUSTIFY rounds up the decimal portion of the value of X.

```
S X=14.445 W $J(X,10,2)
     14.45
```

The following example justifies the value of FRAC in a 10-character field. Because FRAC is a fraction without a leading zero, $JUSTIFY appends a zero to the left of the decimal point before calculating the number of spaces required to create the specified field length.

```
S FRAC=.031 W $J(FRAC,10,3)
     0.031
```

The following example uses $JUSTIFY for format control.

```
PAY   ;PAY=1 CENT A DAY ON DAY 1-THEN DOUBLES
      S PAY=.01
      F DAY=1:1:31 W $J(DAY,2),$J(PAY,14,2),! S PAY=PAY_*2
```

$LENGTH

PURPOSE:

$LENGTH returns either the number of characters in a specified string or the number of substrings in the specified string.

FORM:

$L{ENGTH}(*string*{,*delimiter*})

where:

string	is the string whose length you want to determine or the string whose number of substrings you want to determine
delimiter	is the character or characters used to separate the substrings within *string*

EXPLANATION:

The one-argument form of $LENGTH returns the number of characters in *string*.

The two-argument form of $LENGTH returns the number of substrings within *string*. $LENGTH returns the number of substrings separated from one another by *delimiter*. This number is always equal to the number of delimiters in the string plus one. If the delimiter itself is a string of several characters, then the number is the number of delimiter strings plus one. Thus, the statement:

```
W $L("A,B,C",",")
```

returns

```
3
```

COMMENTS:

Keep the following points in mind when you use the $LENGTH function:

1. If *string* is null in the one-argument form, $LENGTH returns a zero.
2. If *string* is null in the two-argument form, $LENGTH returns a one.
3. If *delimiter* is null, $LENGTH returns zero.

$LENGTH (Cont.)

RELATED:

None.

EXAMPLES:

The following example determines the length of a local variable.

```
S A="TEST" W $L(A)
4
```

The following example determines the number of substrings within a string.

```
S STR="ABC$DEF$EFG",DELIM="$" W $L(STR,DELIM)
3
```

The following example returns a zero because the string tested is the null string.

```
S A="" W $L(A)
0
```

The following example centers the text stored in the variable TXT within a field of spaces. The variable FLD stores the size of the field.

```
        S FLD=21,TXT="HELLO WORLD"
        W "|" D CENTER W TXT,?$X+((FLD-$L(TXT))-TAB),"|" Q
CENTER  S ILEN=$L(TXT),TAB=(FLD-ILEN)\2 F I=1:1:TAB W " "
        Q
```

$NEXT

PURPOSE:

$NEXT returns the subscript of the next sibling in collating sequence to the specified global or local node.

FORM:

$N{EXT}(*storage ref*)

where:

storage ref is the name of a local or global variable or is an indirect reference that evaluates to a variable name or to another indirect reference

EXPLANATION:

You must use only subscripted variable names as arguments. If you are using a naked reference, the naked indicator must be defined.

If a sibling with a higher subscript exists, $NEXT returns its subscript. If no such sibling exists, $NEXT returns a -1.

COMMENTS:

Keep the following points in mind when you use $NEXT:

1. $NEXT returns subscripts for any global variable in the collating sequence in which the global was created. However, $NEXT returns the subscripts of local variables in numeric collating sequence.

2. $NEXT is similar to $ORDER, except that they have different starting values and failure codes, as follows:

	$NEXT	$ORDER,$ZSORT
starting point -1		null string
failure code -1		null string

   In general, $ORDER should be used instead of $NEXT, since $NEXT cannot handle negative subscripts.

$NEXT (Cont.)

RELATED:

The $ORDER Function
The $ZORDER Function
The $ZNEXT Function
The $ZSORT Function
The $ZORDER Special Variable
Array Structure (Section 2.4.4)

EXAMPLES:

The following example returns the first subscript in ^X. It sets the naked indicator to ^X("").

```
S ^X(1,2,3)="1",^X(2)="2"
W $N(^X(-1))
1
```

The following example returns the next subscript on the single-subscripted level. (The node you specify in the argument need not exist.) The naked indicator is still set to ^X("").

```
S ^X(1,2,3)="1",^X(2)="2"
W $N(^X(1))
2
```

The following example returns the first subscript on the two-subscript level. The naked indicator is now set at the ^X(1....) level.

```
S ^X(1,2,3)="1",^X(2)="2"
W $N(^X(1,-1))
2
```

$ORDER

PURPOSE:

$ORDER returns the subscript of the next sibling in collating sequence of a specified array node.

FORM:

$O{RDER}(*storage ref*)

where:

storage ref is the name and subscript(s) of a local or global variable or is an indirect reference that evaluates to a variable name or to another indirect reference

EXPLANATION:

If another sibling exists, $ORDER returns its subscript. If no sibling exists or if the variable specified does not exist, $ORDER returns the null string.

The node you specify in the argument must have at least one subscript. If you are using a naked reference, the naked indicator must be defined.

COMMENTS:

Keep the following points in mind when you use $ORDER:

1. $ORDER returns subscripts for any global variable in the collating sequence in which the global was created. However, $ORDER returns the subscripts of local variables in numeric collating sequence.

2. $ORDER is similar to $ZSORT and $NEXT. All three functions allow you to examine the structure of an array. However, they have different starting points and failure codes, as follows:

	$NEXT	$ORDER,$ZSORT
starting point	-1	null string
failure code	-1	null string

   In all other respects, $ORDER and $NEXT perform identical operations (including the way they return subscripts of local and global variables).

3. $ORDER and $ZSORT perform identical operations on global variables and have identical syntax, but return subscripts of local variables differently. $ORDER returns local variables in numeric collating sequence. $ZSORT returns local variables in ASCII collating sequence.

   For example, if A is local array with the following defined nodes: A(1), A(2), A(2.5), A(3), A(10), A(20), A(01), A("B"), then $ORDER and $ZSORT return the subscripts in the following order:

$ORDER	$ZSORT
$O(A(""))=1	$ZS(A(""))="01"
$O(A(1))=2	$ZS(A(01))=1
$O(A(2))=2.5	$ZS(A(1))=10
$O(A(2.5))=3	$ZS(A(10))=2
$O(A(3))=10	$ZS(A(2))=2.5
$O(A(10))=20	$ZS(A(2.5))=20
$O(A(20))="01"	$ZS(A(20))=3
$O(A(01))="B"	$ZS(A(3))="B"
$O(A("B"))=""	$ZS(A("B"))=""

RELATED:

The $DATA Function
The $NEXT Function
The $ZNEXT Function
The $ZORDER Function
The $ZSORT Function
The $ZORDER Special Variable
Array Structure (Section 2.4.4)

EXAMPLES:

The following example returns the next subscript at the single-subscript level.

```
S ^PR(1)=5549
S ^PR(1.6)=2316
S ^PR(1.8)=647
S ^PR(1.22)="END"
W $O(^PR(1.6))
1.8
```

$ORDER (Cont.)

The following example also returns the next subscript at the single-subscript level.

```
S ASG("ZERO")=10
S ASG("ONE")=11
S ASG("FIVE")=12
S ASG("SIX")=13
W $O(ASG(""))
FIVE

W $O(ASG("FIVE"))
ONE

W $O(ASG("ONE"))
SIX

W $O(ASG("SIX"))
ZERO

W $O(ASG(("ZERO"))
```

The following example returns all subscripts at the single-subscript level from the array TREE. It also returns the value of the node having each subscript.

```
PRINT   S Y=""
STOP    S Y=$O(TREE(Y)) I Y="" Q
        W !,Y,?15,TREE(Y) G STOP
```

$PIECE

PURPOSE:

$PIECE returns the specified substring from the specified string.

FORM:

$P{IECE}(*string,delimiter*{,*start field*{,*end field*}})

where:

string	is the string that contains the substring
delimiter	is the character or characters used as a delimiter between the substrings in the string
start field	is an integer expression that specifies the field (or the first of a series of substrings) you want to retrieve from string
end field	is an integer expression that specifies the last in a series of substrings to retrieve from the string

EXPLANATION:

Each string can be composed of a series of substrings separated by a common delimiter. The delimiter can be any character or series of characters that occur in the string.

All other characters in the string can have a position value with respect to the delimiter. For example, consider the string:

"123#456#789"

If the pound sign (#) is the delimiter, the characters 123 form the first substring and the characters 789 form the third substring.

The two-argument form of $PIECE returns the substring located before the *first* occurrence of *delimiter*. That is, $PIECE returns the first substring in the string.

Thus, the statement:

```
W $P("123#456#789","#")
```

returns the substring:

```
123
```

$PIECE (Cont.)

The three-argument form of $PIECE returns the *start field* substring in the string. That is, $PIECE returns the substring located between the *start field*-1 and *start field* occurrences of *delimiter*.

The four-argument form of $PIECE returns the substring of *string* bounded on the left but not including the *start field*-1 occurrence of *delimiter* in *string*, and bounded on the right but not including the *end field* + 1 occurrence of *delimiter*. The substring that $PIECE returns includes any intermediate occurrences of *delimiter*.

Thus, the statement:

```
W $P("123#456#789","#",1,2)
```

returns the substring

```
123#456
```

COMMENTS:

Keep the following points in mind when you use the $PIECE function:

1. For the two-argument form:

   The two-argument form is equivalent to the three-argument form when the value of *start field* is one.

2. For the three-argument form:

   1. If *start field* is less than one, $PIECE returns a null string.

   2. If *start field*-1 is greater than the number of delimiting substrings, $PIECE returns a null.

3. For the four-argument form:

   1. If *start field* is less than one, $PIECE treats *start field* as having a value of one.

   2. If *end field* is greater than the number of occurrences of *delimiter* in the string, $PIECE returns all characters from those to the left of the *start field* occurrence of *delimiter* through the end of the string.

   3. If *start field* is greater than *end field*, $PIECE returns a null string.

   4. If *end field* is less than one, $PIECE returns a null string.

$PIECE (Cont.)

4. For both the three- and four-argument forms:

    1. If there are fewer than *start field*-1 instances of *delimiter* in *string*, $PIECE returns a null string.

    2. If the substring specified in *delimiter* does not exist in *string* and if *start field* is one, $PIECE returns the entire string unless *end field* equals zero.

    3. If *delimiter* is a null string, $PIECE returns a null string. (Null substrings can be found anywhere in any number required.)

5. For all forms:

    To set the value of a substring within a string you can use an expression that resembles the $PIECE function as an argument of the SET command. See the description of the SET command for details about this process.

RELATED:

The SET Command
The $EXTRACT Function
The $FIND Function
The $LENGTH Function

EXAMPLES:

The following example shows that the two-argument form is equivalent to the three-argument form when *start field* is one.

```
S A="123#456#789"
W $P(A,"#",1),!,$P(A,"#")
123
123
```

The following example finds the third substring in string A.

```
S A="123#-#456#-#789"
W $P(A,"#-#",3)
789
```

The following example returns all substrings from that immediately to the left of the third occurrence of *delimiter* through that immediately to the left of the fifth occurrence of *delimiter*.

```
W $P("JAN;FEB;MARCH;APR;MAY;JUN",";",3,5)
MARCH;APR;MAY
```

$PIECE (Cont.)

The following example extracts the fractional portion of the result of an arithmetic operation.

```
W $P(355/113,".",2)
14159292035
```

The following example shows that you can nest $PIECE. It finds the second piece of A marked by the "^" delimiter and the first and second pieces of this substring (A,B,C) marked by the "," delimiter.

```
S A="1,2,3^A,B,C^@#!"
W $P($P(A,"^",2),",",1,2)
A,B
```

$RANDOM

PURPOSE:

$RANDOM returns a pseudorandom integer uniformly distributed in a closed interval from zero through one less than the specified integer value.

FORM:

$R{ANDOM}(*integer*)

where:

integer is an integer-valued expression one greater than the largest pseudorandom integer that you want $RANDOM to return

EXPLANATION:

The value of *integer* must be nonzero.

RELATED:

None.

EXAMPLES:

The following example always returns a zero.

```
W $R(1)
0
```

The following example generates an array of evenly distributed random numbers from zero through 24.

```
F I=1:1:50 S X(I)=$R(25)
```

The following example always generates an even integer between two and 102.

```
S X=$R(51)+1*2
W X
64
```

The following example simulates the roll of two dice.

```
DICE   R !,"ROLL DICE ?",A Q:A=""
       W !,$R(6)+1,"+",$R(6)+1 G DICE
```

$SELECT

PURPOSE:

$SELECT returns the value of the first (leftmost) expression in its argument list whose matched truth-valued expression is true.

FORM:

$S{ELECT}(*truth value:expression*{,...})

where:

truth value is a truth-valued expression

expression is an expression

EXPLANATION:

Each $SELECT argument is a pair of expressions separated by a colon. The left half of the pair is a truth-valued expression. The right half of the pair can be any expression.

$SELECT evaluates the arguments from left to right. When $SELECT discovers a truth-valued expression with the value of one (true), it returns that truth-valued expression's matching right expression.

COMMENTS:

Keep the following points in mind when you use the $SELECT function:

1. $SELECT stops evaluation after it discovers the leftmost true truth-valued expression. It never evaluates later pairs.

2. If all truth-valued expressions in the argument list are false, DSM-11 produces an error. To prevent an error from disrupting an executing routine, make sure that one of the pairs evaluates as true.

   (See the *DSM User's Guide* for more information on errors.)

EXAMPLES:

The following example prompts for a function and calls a routine based on the answer. If none of the expected answers are received, DSM-11 invokes the error-handling routine ERR.

```
R !,"FUNCTION 1,2, OR 3?",A
D @$S(A=1:"^TEST",A=2:"^UPDAT",A=3:"^EDI",1:"^ERR")
```

$SELECT (Cont.)

The following example accepts a number and determines if the number is odd or even.

```
A   R !,"ENTER A NUMBER",X Q:X=""
    W !," THE NUMBER IS ",$S(X#2:ODD,1:"EVEN")
    G A
```

$TEXT

PURPOSE:

$TEXT returns the specified line from the routine currently in memory.

FORMS:

$T{EXT}(*argument*)

In which *argument* can have any of the following forms:

line ref

line spec

where:

line ref	is a line reference to the line in the routine or is an indirect reference that evaluates to a line reference or to another indirect reference
line spec	is a line specification of the line in the routine

EXPLANATION:

$TEXT returns the line referenced from the current routine in memory in a special format. $TEXT replaces the TAB character with a space character. $TEXT does not return the carriage return that terminates the line.

COMMENTS:

Keep the following points in mind when you use the $TEXT function:

1. If the line reference or line specification is to a line that does not appear in the body of the routine, $TEXT returns a null string.

2. If you want to return the name of the routine, use a line specification of +0.

RELATED:

The ZINSERT Command
The ZREMOVE Command
Line References (Section 2,5,1)
Line Specifications (Section 2.5.3)

$TEXT (Cont.)

EXAMPLES:

The examples are based on the following simple routine named MATH:

```
MATH   ;MATHEMATICAL ROUTINE
       W "THIS ROUTINE MANIPULATES TWO NUMBERS"
       W !,"A CARRIAGE RETURN EXITS"
STAR   R !!,"ENTER NUMBER ",X Q:X=""
SEC R  !!,"ENTER SMALLER NUMBER ",Y
       I Y'<X G SEC
       W !!,"THE NUMBERS ADDED = ",X+Y
       W !!,"THE NUMBERS SUBTRACTED =",X-Y
       W !!,"THE NUMBERS MULTIPLIED = ",X*Y
       W !!,"THE NUMBERS DIVIDED = ",X/Y
       G STAR
```

The following example uses a line label to retrieve the line labeled STAR.

```
W $T(STAR)
STAR R !!,"ENTER NUMBER ",X Q:X=""
```

The following example uses an offset to retrieve the line beyond that labeled SEC. (The first character on the line that DSM-11 returns is a space.)

```
W $T(SEC+1)
 I Y'<X G SEC
```

The following example uses a line specification to retrieve the third line in the routine. (The first character on the line that DSM-11 returns is a space.)

```
W $T(+3)
 W !,"A CARRIAGE RETURN EXITS"
```

The following example returns the name of the routine.

```
W $T(+0)
MATH
```

$VIEW

PURPOSE:

$VIEW returns the decimal equivalent of the contents of a specified memory location.

FORMS:

$V{IEW}(*location*{,*state*})

where:

location	is an integer-valued expression specifying a location in memory
state	is an integer-valued expression specifying the exact meaning of location

EXPLANATION:

The one-argument form of $VIEW reads a location in memory specified by *location*. The value of *location* must be the logical address used in Kernel-Mode mapping.

The two-argument form of $VIEW performs several different viewing tasks. DSM-11 determines which viewing task to perform by the value in the second argument.

Table 6-1 gives the possible values for *state* and the meanings they give *location*.

$VIEW (Cont.)

Table 6-1: $VIEW Argument Values

location value *value*	*state* Value	Meaning
0 - m	1 - n	*state* specifies a job number. (n stands for the maximum number of jobs on the system.) *location* specifies an offset from the beginning of the partition. (m stands for the largest offset value possible in the partition.) To specify your own job, use the special variable $JOB for *state*.
0 - x	0	*state* specifies that *location* is an offset from the beginning of the VIEW Buffer. (x stands for the largest offset value on the VIEW device.)
0 - 65535	-1	*state* specifies that *location* is the logical address used in Kernel-Mode mapping. (This is the default *state* when you do not include a *state* specifier.)
0 - 65535	-2	*state* specifies that *location* is the logical address used in User-Mode mapping.
0 - 16383	+128 to +32767	*state* specifies a memory-management block.*location* is an offset starting at the memory-management block specified by *state*.

When the *state* value is either -1 or -2, a *location* of 40960 through 56344 represents the current partition. A *location* value of 56344 to 65535 represents the I/O page.

When you use a *state* value between +128 and +32767, you can view an area in memory that would otherwise not be mapped by either Kernel Mode (*state* = -1) or User Mode (*state* = -2) at the time when you execute the VIEW. Thus, these *state*values allow you to view anywhere into physical memory, except for the first 8192 bytes, which are accessible as either Kernel Mode or User Mode.

To obtain a more detailed explanation, see your *PDP-11 Processor Handbook* for information on memory management.

$VIEW (Cont.)

COMMENTS:

Keep the following points in mind when you use the $VIEW function:

1. You can use either bytes or full words to specify *location*. If *location* is an even number, it specifies a word *location* and $VIEW returns a 16-bit word value. If *location* is an odd number, it specifies an odd byte *location* and $VIEW only returns the value of that 8-bit byte.

   (To obtain the value of any byte whose address may be even or odd, use the formula $V(...)#256.)

2. You may find $VIEW most useful if you are writing utility routines that must read the system tables. See the *DSM-11 User's Guide* for more information on system tables and $VIEW.

RELATED:

The VIEW Command

EXAMPLES:

The following example examines the high byte of word 2.

```
W $VIEW(3)
```

The following example examines Kernel-Mode location 42. This word consists of bytes 42 and 43.

```
S X=$V(42)
```

The following example examines location 50 in partition number 2.

```
S WD=$V(50,2)
```

The following example examines the last word in the VIEW buffer.

```
S WD=$V(1022,0)
```

$ZCALL

PURPOSE:

$ZCALL provides an general-purpose function call to user-written routines.

FORM:

$ZC{ALL}(*external ref*{,*expression*,...})

where:

external ref is a symbol defined outside the DSM-11 language

expression is an expression

EXPLANATION:

Your System Manager or system programmer must previously have established the actual spelling and function of the symbol represented by *external ref*. That spelling must conform to the syntax rules for DSM names. Thus, *external ref* must contain no more than eight uppercase alphabetic or digit characters. The first character must be either an alphabetic or a percent character (%).

Your System Manager or system programmer must also have determined the number and nature of the expressions that follow *external ref*. These expressions must relate to *external ref* as arguments relate to a function.

COMMENTS:

$ZCALL gives you a consistent method for specifying functions unique to your site. By using $ZCALL, you do not have to add new $Z functions or make extensive modifications to the interpreter.

Whether or not you can use $ZCALL depends on its implementation at your site. See the *DSM User's Guide* for more information on using $ZCALL.

RELATED:

None.

EXAMPLES:

None.

$ZNEXT

PURPOSE:

$ZNEXT performs a physical scan of an array.

FORM:

$ZN{EXT}(*global ref*)

where:

global ref	is the name and subscript(s) of global variable or is an indirect reference that evaluates to a global name or to another indirect reference

EXPLANATION:

$ZNEXT returns a full reference (name and subscripts) to the next defined node in collating sequence to the node you specify in the argument. If no such node exists, $ZNEXT returns a -1.

COMMENTS:

$ZNEXT and $ZORDER perform similar operations, that is, they allow you to examine the structure of a global array. However, $ZNEXT and $ZORDER have two syntactic differences: starting points and failure codes. These values are shown in the following chart:

$ZNEXT	$ZORDER
starting point -1	null string
failure code -1	null string

Refer to the description of $ZORDER for details about the method these two functions use to determine array structure. In general, $ZORDER should be used instead of $ZNEXT.

Both $ZNEXT and $ZORDER work as indicated only on global variables. If you try to use these functions on a local subscripted variable, you receive a <SYNTX> error.

$ZNEXT (Cont.)

RELATED:

The $DATA Function
The $NEXT Function
The $ORDER Function
The $ZORDER Function
The $ZSORT Function
The $ZORDER Special Variable
Array Structure (Section 2.4.4)
Naked References (Section 2.4.5)

EXAMPLES:

The following example writes all the defined nodes in the array X.

```
S ^X(1,2,3)="1",^X(2)="2",^X(1,2,3,4)="12"
W $ZN(^X(""))
^X(1,2,3)

W $ZN(^X(1,2,3))
^X(1,2,3,4)

W $ZN(^X(1,2,3,4))
^X(2)
```

The following example produces a list of all defined nodes in the global variable ^A.

```
A   S X=$ZN(^A(-1))
    F I=1:1 Q:X=-1  W !,X,"=",@X S X=$ZN(@X)
```

$ZORDER

PURPOSE:

$ZORDER performs a physical scan of an array.

FORM:

$ZO{RDER}{*global ref*}

where:

global ref	is the name and subscript(s) of a global variable or is an indirect reference that evaluates to a global name or to another indirect reference

EXPLANATION:

$ZORDER returns a full reference (name and subscripts) to the next defined global node in collating sequence to the global node you specify in the argument. If no such global node exists, $ZORDER returns a null string.

COMMENTS:

Keep the following points in mind as you use the $ZORDER function:

1. $ZORDER and $ZNEXT perform similar operations; that is, they allow you to determine the structure of a global array. However, $ZORDER and $ZNEXT have two syntactic difference: their starting points and failure codes, as follows:

	$ZORDER	$ZNEXT
starting point	null string	-1
failure code	null string	-1

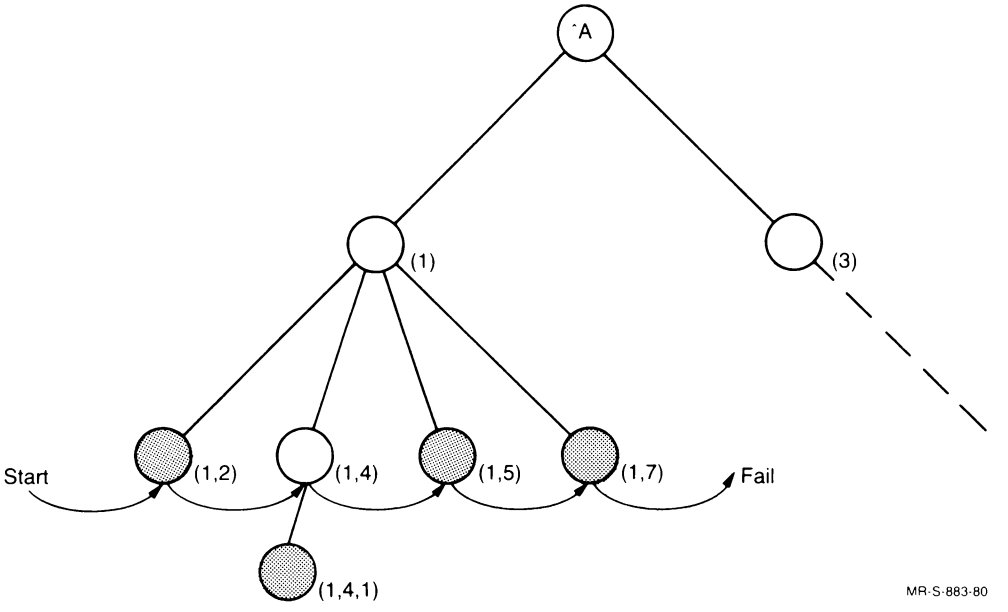
2. Both $ZNEXT and $ZORDER give the results indicated here only on global variables. If you try to use them on a local subscripted variable, you receive a SYNTX error.

3. $ZORDER $NEXT, $ZNEXT, $ORDER, and $ZSORT can all be considered trace functions; they allow you to examine the form and contents of an array. However, the method $ORDER, $ZSORT, and $NEXT use is different from the method $ZORDER and $ZNEXT use.

$ZORDER (Cont.)

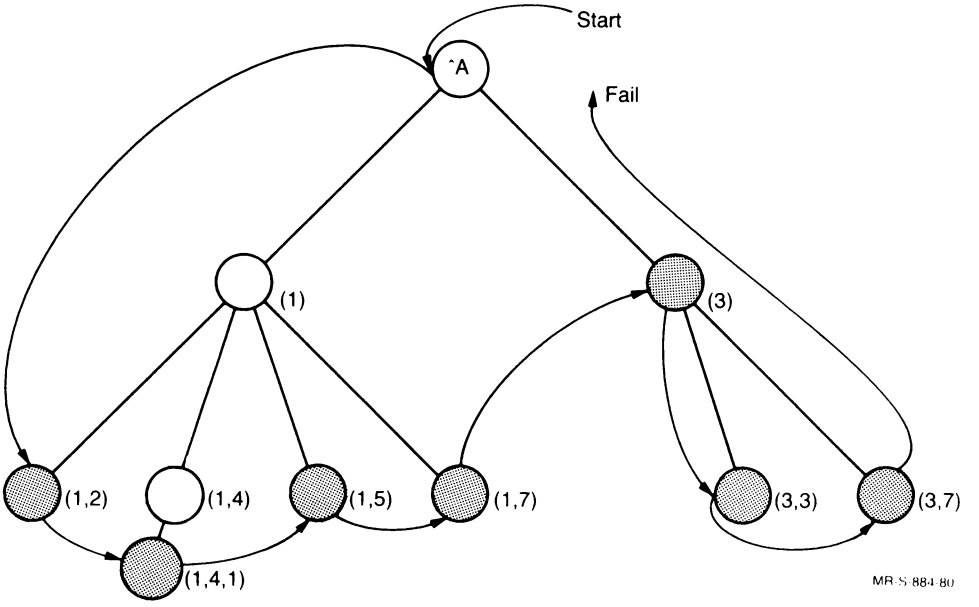
$ORDER, $ZSORT, and $NEXT return only the subscripts of siblings (nodes having a common parent); but they return both data (defined) nodes and logical (pointer) nodes. Figure 6-1 shows the method $ORDER, $ZSORT, and $NEXT use. (All nodes filled in black are data nodes. All nodes not filled in black are pointer nodes.)

Figure 6-1: A $ORDER, $ZSORT, or $NEXT Array Scan



On the other hand, $ZORDER and $ZNEXT return all data nodes in an array, regardless of their relationship; but they do not return pointer nodes. Figure 6-2 shows the method $ZORDER and $ZNEXT use. (All nodes filled in black are data nodes. All nodes not filled in black are pointer nodes.)

Figure 6-2: A $ZORDER, or $ZNEXT Array Scan



For example, if the global array contains the following defined nodes:

^A(1)
^A(1,1)
^A(1,3)
^A(3)
^A(3,2)
^A(3,4,5)

$ZORDER (Cont.)

$NEXT/$ORDER/$ZSORT and $ZORDER/$ZNEXT return the following information:

	$NEXT/$ORDER/$ZSORT	$ZORDER/$ZNEXT
Argument	Return	Return
^A(1)	3	^A(1,1)
^A(1,1)	3	^A(1,3)
^A(1,3)	Fail	^A(3)
^A(3)	Fail	^A(3,2)
^A(3,2)	4	^A(3,4,5)
^A(3,4,5)	Fail	Fail

RELATED:

The $DATA Function
The $NEXT Function
The $ORDER Function
The $ZNEXT Function
The $ZSORT Function
The $ZORDER Special Variable
Array Structure (Section 2.4.4)
Naked References (Section 2.4.5)

EXAMPLES:

The following example writes defined nodes for the array ^X:

```
S ^X(1,2,3)=1,^X(2)=2,^X(1,2,3,4)=12
W $ZO(^X(""))
^X(1,2,3)

W $ZO(^X(1,2,3))
^X(1,2,3,4)

W $ZO(^X(1,2,3,4))
^X(2)
```

$ZSORT

PURPOSE:

$ZSORT returns the subscript of the next sibling in collating sequence of a specified array node.

FORM:

$ZS{ORT}(*storage ref*)

where:

storage ref	is the name and subscript(s) of a local or global variable or is an indirect reference that evaluates to a variable name or to another indirect reference

EXPLANATION:

If another sibling exists, $ZSORT returns its subscript. If no sibling exists or if the variable specified does not exist, $ZSORT returns the null string.

The node you specify in the argument must have at least one subscript. If you are using a naked reference, the naked indicator must be defined.

COMMENTS:

$ZSORT is similar to $ORDER. Both allow you to examine the structure of an array. They have identical starting points and failure codes (null string). They both return the subscripts of any global variable in the collating sequence in which the global was created. However, $ZSORT and $ORDER return subscripts of local variables differently. $ZSORT returns local variables in ASCII collating sequence. $ORDER returns local variables in numeric collating sequence.

For example, if A is a local array with the following defined nodes: A(1), A(2), A(2.5), A(3), A(10), A(20), A(01), A("B"), $ORDER and $ZSORT return the subscripts in the following order:

$ORDER	$ZSORT
$O(A("")) = 1	$ZS(A("")) = 01
$O(A(1)) = 2	$ZS(A(01)) = 1
$O(A(2)) = 2.5	$ZS(A(1)) = 10
$O(A(2.5)) = 3	$ZS(A(10)) = 2
$O(A(3)) = 10	$ZS(A(2)) = 2.5
$O(A(10) = 20	$ZS(A(2.5)) = 20
$O(A(20)) = 01	$ZS(A(20)) = 3
$O(A(01)) = "B"	$ZS(A(3)) = "B"
$O(A("B")) = ""	$ZS(A("B")) = ""

$ZSORT (Cont.)

RELATED:

The $DATA Function
The $NEXT Function
The $ORDER Function
The $ZORDER Function
The $ZNEXT Function
The $ZORDER Special Variable
Array Structure (Section 2.4.3)

EXAMPLE:

The following example returns the next subscript at the single-subscript level.

```
S ^PR(1)=5549
S ^PR(1.6)=2316
S ^PR(1.8)=647
S ^PR(1.22)="END"

W $ZS(^PR(1.6))
1.8
```

$ZUCI

PURPOSE:

The $ZUCI function can be used to obtain the three-letter UCI name given a UCI number, or the UCI number if given the UCI name.

FORM:

$ZU{CI}(*name*{,*volume set name*})

or

$ZU{CI}(*number*,*volume set number*})

where:

name	is a three-letter code or a null string
volume set name	is a three-letter code
number	is 0 or a positive integer from 1 to 50
volume set number	is 0 or a positive integer from 1 to 3

EXPLANATION:

$ZU can be used to find out the current UCI name or number, or to find out UCI names or numbers for other UCIs. The $ZU function returns a UCI name or number along with the name or number of the volume set the UCI resides on.

The $ZU function works as follows:

Input	Result
$ZU(UCI name)	UCI number, volume set number
$ZU(UCI number)	UCI name, volume set name
$ZU("")	Current UCI number, volume set number
$ZU(0)	Current UCI name, volume set name

$ZUCI (Cont.)

If you want to find out information about a UCI on a different volume set from the one you are currently on, then you must specify the volume set, for example:

$ZU(*name*,*volume set name*)

or:

$ZU(*number*,*volume set number*)

COMMENTS:

If a nonexistent UCI name, UCI number, volume set name, or volume set number is submitted, a <NOUCI> error will result.

RELATED:

None

EXAMPLES:

The account SCT has a UCI number of 11, and the system volume set has the name SYS and a volume set number of 0.

```
WRITE $ZU("SCT","SYS")
11,0

WRITE $ZU(11,0)
SCT,SYS
```

Chapter 7
DSM-11 Special Variables

This chapter describes the DSM-11 special variables and provides examples of their use.

7.1 Introduction to DSM-11 Special Variables

Special variables are system-defined and maintained variables that contain information on various values or processes in the operating environment. During the course of processing, the system updates information stored in the special variables. If you need to know or use the information a special variable contains, you can access it by using the special variable name.

Each special variable is an uppercase name preceded by a dollar sign. The name is a mnemonic for the information the special variable contains.

DSM-11 has two types of special variable:

- ANSI Standard special variables
- Extended special variables

ANSI Standard special variables are specified in the ANSI Standard and follow standard usage. The *ANSI MUMPS Language Standard* reserves the characters $A through $Y as the first two characters of Standard special variable mnemonics.

As specified in the ANSI Standard, extended special variables are implementation-specific additions to the language. Extended special variables use the characters $Z as the first two characters of their mnemonics.

7.2 Special Variable Descriptions

The following pages contain descriptions of all DSM-11 special variables. Each description contains the purpose, form, and operation of the variable. The descriptions are in alphabetical order for ease of referencing.

Each description also contains one or more examples of how to use the special variable. All command-line examples in the following descriptions are given as they would appear when entered from a terminal. All routine-line examples are given as they would appear when listed on a line printer.

$HOROLOG

PURPOSE:

$HOROLOG contains two integer values that are the current time and date.

FORM:

$H{OROLOG}

EXPLANATION:

$HOROLOG returns the time and date in the following form:

date,time

The system increments the time field from zero to 86399 seconds. When it reaches 86399 at midnight, the system resets the time field to zero and increments the date field by one.

COMMENTS:

DSM-11 keeps a time and date based on the premise that day zero is December 31, 1840. Thus $HOROLOG contained a value of 0,1 at 12:00:01 on December 31, 1840.

RELATED:

None.

EXAMPLE:

The following example converts $HOROLOG to a time in the form hour:minute AM or PM.

```
10    S M=$P($H,",",2)\60
      S N="AM" S:M'<720 M=M-720,N=" PM"
      S:M<60 M=M+720
      S I=M\600 S:'I I=""
      S TIM=I+(M\60#10)_":"_(M#60\10)_(M#10)_N
      I '$D(NP) W TIM K TIM
      K M,N,I,NP Q
INT   S NP="" G 10
```

$IO

PURPOSE:

$IO contains the specifier of the current I/O device.

FORM:

$I{O}

EXPLANATION:

Whenever you issue a USE command to direct I/O to a device, DSM-11 sets the value of $IO to a specifier of that device. Whenever you issue a CLOSE command to give up ownership of the device and return to your principal device, DSM-11 sets the value of $IO to that of your principal device.

RELATED:

None.

EXAMPLES:

The following example uses a device (specified in the variable DEV), returns to the principal device, and writes the device specification to the device.

```
OP1     O DEV U DEV S I=$I
        U 0 W I,!
```

The following example saves the current device specification in OLDI, opens the specified device, and prints the symbol table. Then it reestablishes the old device specification saved in OLDI.

```
BEGIO   R !,"ENTER DEVICE TO USE ",DEV
        S OLDI=$I
        O DEV U DEV ZW    C DEV
        U OLDI
```

$JOB

PURPOSE:

$JOB contains the integer job number assigned to your job.

FORM:

$J{OB}

EXPLANATION:

Every active DSM-11 job has a unique job number. (See the *DSM-11 User's Guide* for more information.)

RELATED:

None

EXAMPLE:

The following example prints a form feed, the job number, the symbol table, and the current routine on a specified device.

```
O DEV U DEV W #,$J,! ZW    W ! ZP
```

$STORAGE

PURPOSE:

$STORAGE contains the amount of free space available within your partition.

FORM:

$S{TORAGE}

EXPLANATION:

$STORAGE contains the amount of free space in bytes.

COMMENTS:

See the *DSM User's Guide* for more information on the partition.

RELATED:

None.

EXAMPLES:

The following example writes the number of bytes available in the current partition.

```
W $S
3783
```

The following example determines the number of bytes available in the work area. It executes a FOR statement to set a local array and determines the amount of space remaining.

```
K
ZR
W $S
5904
F I=1:1:100 S A(I)=I
W $S
4906
```

$TEST

PURPOSE:

$TEST contains the truth value resulting from the execution of an IF command with an argument, an OPEN, LOCK, or READ command containing a timeout, or a JOB command.

FORM:

$T{EST}

EXPLANATION:

DSM-11 sets $TEST in three situations:

1. DSM-11 assigns $TEST the truth value from the most recently executed IF command containing an argument.

   If the IF command argument is true, DSM-11 assigns $TEST the value of true (one). If the IF command argument is false, DSM-11 assigns $TEST the value of false (zero).

2. DSM-11 assigns $TEST a truth value from the most recently executed LOCK, OPEN, or READ command containing a timeout.

   If the operation specified by the command is completed before the timeout occurs, DSM-11 enters a truth value of true (one). If the operation specified by the command is not completed before the timeout occurs, DSM-11 enters a truth value of false (zero).

3. DSM-11 assigns $TEST a truth value from the most recently executed JOB command.

   If the job started successfully, DSM-11 enters a truth value of true (one). If the job did not start successfully (that is, if DSM-11 has no partitions available, or has no partitions of the specified size or greater available), DSM-11 enters a truth value of false (zero).

COMMENTS:

Keep the following points in mind when using $TEST:

1. Postconditional expressions do not set $TEST.

$TEST (Cont.)

2. The special variable $TEST is not stacked by the DO command (with arguments). This means that called subroutines can alter the value of $TEST unexpectedly, for example:

```
ROUT    I A>5 DO ^BIG
        E    DO ^LITTLE
```

   This routine fragment may not work as expected if the value of $TEST is altered in the routine ^BIG.

3. The value of $TEST is stacked by an argumentless DO command, and restored by the QUIT that ends the execution block.

RELATED:

The DO Command
The IF Command
The LOCK Command
The OPEN Command
The READ Command
The JOB Command
Postconditional Expressions (Section 2.7.4)
Timeout Expressions (Section 2.7.5)
Truth-Valued Expressions (Section 2.7.3)

EXAMPLE:

The following example uses $TEST with a timed READ. If the answer is entered fully before the timeout, the example writes the complete reply. If the answer is not entered fully before the timeout, the example writes the incomplete reply.

```
T    R !,"ANSWER IN TEN SECONDS:",ANS:10
     W !,$S($T:"ANSWER =",1:"PARTIAL ANS. =")),ANS
     G T
```

$X

PURPOSE:

$X contains a nonnegative integer whose value is the horizontal carriage or cursor position on the current line.

FORM:

$X

EXPLANATION:

$X can have a value of from zero through 255. Each READ or WRITE statement adds the length of the string read or written. The READ * and WRITE * forms have no effect on $X.

DSM-11 sets $X to zero whenever it encounters a new-line specifier (a carriage return on a terminal) or whenever the count in $X exceeds 255. Thus, DSM-11 can increment $X from 255 to zero for additions that would bring the count above 255, but it cannot decrement $X from 0 to 255 for deletions that would bring the count below zero.

COMMENTS:

The following formatting characters also affect $X:

! (new line operator) sets $X to zero

(form feed) sets $X to zero

?n (horizontal tabulation) sets $X to the larger of the two values: n or the current value in $X

RELATED:

Formatting Characters (Section 2.6.7)

EXAMPLE:

The following example writes a 7-character string. Then, it gives the variable P the current value in $X and writes P.

```
W !,"TESTING" S P=$X W !,P
TESTING
7
```

$Y

PURPOSE:

$Y contains a nonnegative integer value that specifies the current line of your I/O device.

FORM:

$Y

EXPLANATION:

The value in $Y always equals the number of new-line operations that have occurred since the last form feed. $Y can have a value between zero and 255. DSM-11 sets $Y to zero whenever it encounters a form feed or whenever the count exceeds 255.

COMMENTS:

The following formatting characters also affect $Y:

! (carriage return/line feed) adds one to $Y

(form feed) reinitializes $Y to zero

The READ * and WRITE * forms have no effect on $Y.

RELATED:

Formatting Characters (Section 2.6.7)

EXAMPLES:

The following example causes a form-feed operation and five new-line operations. Then, it gives the current value of $Y, which is 5.

```
W #!!!!  W $Y


5
```

The following example code fragment skips to a new page, tabs 20 characters to the right, and prints the title BILLING REPORT if the number of lines on the current page exceeds 60.

```
TITL W:$Y)60 #?20,"BILLING REPORT"
```

$ZA

PURPOSE:

$ZA contains status or error information for the current device.

FORM:

$ZA

EXPLANATION:

$ZA contains an integer. The meaning of the integer depends on the current device. See the discussion of I/O devices in the *DSM-11 User's Guide* for a description of these devices and the values that $ZA can contain for them.

COMMENTS:

$ZA is useful in examining I/O device status, including errors. You should examine its contents after each I/O operation (to the relevant devices) to determine if an error occurred.

RELATED:

The $ZB Special Variable

EXAMPLES:

None.

$ZB

PURPOSE:

$ZB can contain status information for the current device.

FORM:

$ZB

EXPLANATION:

$ZB contains a numeric value. The meaning of that value depends on the current device. See the discussion of I/O devices in the *DSM-11 User's Guide* for a description of these devices and the values $ZB can contain for them.

DSM-11 distinguishes the abbreviation for $ZBREAK, $ZB, from the special variable $ZB by its context in the routine. You can set the value of $ZBREAK with the SET command; however, you cannot set $ZB. Thus, DSM-11 never confuses the two.

RELATED:

The $ZA Special Variable

EXAMPLE:

The following example uses $ZB to report on the status of a JOB command.

```
STRT    J ^WATCH
        I $T=0 W !,"NO JOB" G REST
        W !,"WATCH STARTED AS JOB#",$ZB
```

$ZBREAK

PURPOSE:

$ZBREAK can contain a reference to a line, a routine, and a command that DSM-11 interprets as a breakpoint when the debugging facility is activated.

FORM:

$ZB{REAK}{(*n*)}

where:

n is an integer from 0 to 9.

EXPLANATION:

$ZBREAK establishes one or more breakpoints in DSM routines. Breakpoints are places in the routine where execution is to be temporarily suspended during a debugging session. A breakpoint specification consists of a full entry reference (line label, optional offset, and routine name) and a command number in the following format:

SET $ZBREAK(*n*) = *entry ref:command number*

The command number is the numerical position of the command within the specified routine line. A colon (:) separates the command number from the entry reference.

To establish a breakpoint, use the SET command to set $ZBREAK to the desired breakpoint specification. For example, the following statement sets a breakpoint at line T in the routine TEST at the first command on the line.

```
SET $ZBREAK="T^TEST:1"
```

Up to nine breakpoints can be in effect at any time. To set multiple breakpoints, or to set a specific breakpoint, use a subscript on $ZBREAK. Valid subscripts are integers between one and nine. For example, the following statement sets breakpoint three. (Note that breakpoints one and two need not exist.)

```
SET $ZBREAK(3)="T^TEST:5"
```

If no subscript is given, DSM-11 sets $ZBREAK(n), where n is the next available subscript in ascending order. If n is greater than nine, then $ZBREAK(9) is set; and the previous $ZBREAK(9) is lost.

$ZBREAK (Cont.)

Optionally, you can set the value of $ZBREAK to a breakpoint and an *action string*. An action string is one or more DSM commands that are executed whenever DSM-11 encounters the breakpoint. The general format of a breakpoint that includes an action string is:

entry ref:command number>action string

An action string can contain any DSM command. A right angle bracket (>) separates the breakpoint specification from the action string. For example, the following statement sets a breakpoint at line STRT + 5 in routine TEST2 at the third command on the line.

```
SET $ZBREAK="STRT+5^TEST2:3)ZW ZG"
```

When the breakpoint is encountered, DSM-11 writes the contents of the local symbol table (allowing you to examine the contents of all variables), and then resumes execution of the routine.

Use $ZBREAK(0) with the SET command to establish an action string that executes each time that you single step in Break Mode. (Although you can also set a breakpoint for $ZBREAK(0), everything to the left of the > is ignored.)

Refer to the *DSM-11 User's Guide* for a detailed explanation of the DSM-11 debugger.

COMMENTS:

Keep the following points in mind when using the $ZBREAK special variable:

1. When DSM-11 encounters a $ZBREAK in a routine (and the debugger is activated), it displays a message on the terminal that is identical to that of the DSM-11 BREAK command, as shown below.

   ```
   <BREAK>entry ref:command number routine-line
   DB>
   ```

2. When you receive the $ZBREAK message, DSM-11 is in the Break Mode. You can execute any DSM-11 command except ZGO with an argument.

   To resume execution of the routine from the $ZBREAK message, issue the ZGO command or the ZBREAK OVER, ZBREAK IN, or ZBREAK OUT command. (You can also type a carriage return in place of ZBREAK IN.) See the ZBREAK command in this manual or the information on the DSM-11 debugger in the *DSM-11 User's Guide*.

3. In Programmer Mode, you can cause a break to occur by pressing CTRL/B. The effect is the same as a BREAK command or as a breakpoint placed immediately after the current command.

$ZBREAK (Cont.)

4. If the debugger is not activated, then breakpoints set by $ZBREAK are ignored. The debugger is turned on by ZBREAK ON and is turned off by ZBREAK OFF.

5. If you set $ZBREAK without a subscript, DSM-11 assigns the breakpoint the first unused breakpoint number between one and nine. If breakpoints one through nine already exist, DSM-11 clears breakpoint nine and assigns the new specification to breakpoint nine.

6. When DSM-11 encounters a breakpoint in a routine, it suspends execution of the routine at the command number of the breakpoint specification before executing that command.

7. DSM-11 distinguishes the abbreviation for $ZBREAK, $ZB, from the special variable $ZB by its context in the routine. You can set the value of $ZBREAK with the SET command; however, you cannot set $ZB. Thus, DSM-11 never confuses the two.

8. You must specify a breakpoint (that is, the full entry reference) in its simplest form. The line label must be the closest label to the line at which you want the break to occur. If the routine is loaded by name, you must include the routine name in the entry reference. If the routine is loaded from a sequential device, omit the routine name.

   For example, in the following program fragment, any breakpoint following the line CALC must include CALC in the label specification:

```
STRT    ;Program TEST1
        S CONSTANT=.75
        F Y=1:1 D CALC Q:Y=100
CALC    S FINAL=CONSTANT/Y
        I FINAL).01 W "Tolerance Unacceptable",!
```

   To specify the breakpoint at the first command on the first line following CALC, the entry reference must be:

   "CALC+1^TEST1:1"

   You cannot specify the line label as:

   STRT+4

9. You cannot set a breakpoint on a routine line that contains no commands (that is, on comment lines).

$ZBREAK (Cont.)

10. You can examine the contents of all existing breakpoints by issuing the command:

    ```
    ZWRITE $ZBREAK
    ```

    You can examine the contents of any specific breakpoints by issuing the following command:

    ```
    WRITE $ZBREAK(n)
    ```

    where *n* is the subscript of the breakpoint that you want to examine.

11. You can delete all existing breakpoints by issuing the command:

    ```
    KILL $ZBREAK
    ```

    You can clear a specific breakpoint by setting the breakpoint to the null string, as follows:

    ```
    S $ZBREAK(2)=""
    ```

12. Each breakpoint takes up 128 bytes of space in your partition. The space is not reclaimed until you issue a KILL $ZBREAK.

RELATED:

The BREAK command
The ZBREAK command
The ZGO command
Entry references (Section 1.6.2)

EXAMPLES:

The following example sets a breakpoint in the routine CALC. The breakpoint occurs just before the DO command that calls the subroutine ITEMS.

```
CALC    ;Calculating squares of even numbers between 1 and 250
        S $ZB(1)="CALC+2^CALC:2"
        F I=1:1 D ITEMS Q:I=250


        .
ITEMS   I I#2=0 S SQ=I*I W !,"The square of I is: ",SQ
        Q
```

$ZERROR

PURPOSE:

$ZERROR contains the last error message and the line label, the routine name, and the command number at which it was generated.

FORM:

$ZE{RROR}

EXPLANATION:

When an error occurs, DSM-11 sets $ZERROR to a string that contains the error message generated and the line label plus offset, routine name, and command number at which the error message was generated.

After setting $ZERROR, DSM-11 transfers control to the line and/or routine reference in the $ZTRAP special variable (if the programmer set $ZTRAP for that level of the routine). For other cases, see the discussion on error handling in the *DSM-11 User's Guide*.

RELATED:

The $ZTRAP Special Variable

EXAMPLE:

The following example writes the contents of $ZERROR after an error. (Line A of routine TEST contained the syntactically incorrect statement W !A,!,B.)

```
W $ZE
A^TEST:1 W !A,!,B
```

See the *DSM User's Guide* for more information on error handling.

$ZORDER

PURPOSE

The $ZO special variable returns the value of the next global node in sequence after the current global reference.

FORM:

$ZO{RDER}

EXPLANATION:

The DSM-11 operating system maintains a current global reference that indicates the last reference to a global. The $ZO variable returns the next physical global node relative to the current global reference. The next reference is determined by the same procedure used by the $ZORDER function. In fact, the following two statements are equivalent:

```
SET X=$ZO

SET X=@($ZO($ZR))
```

where $ZR is a special variable containing the current global reference.

COMMENTS:

If the last reference to a global was a KILL or produced an error, then use of the $ZORDER variable produces an <UNDEF> error.

The $ZORDER variable can be used to print out the values in a global. The values can be printed without $ZORDER, but at a cost of greater execution time.

RELATED:

The $ZORDER Function
The $ZREFERENCE Special Variable
Array Structure (Section 2.4.4)

EXAMPLE:

Global A contains the following nodes and values:

^A(1,1) = 10
^A(1,2) = 20
^A(1,3) = 30
^A(1,4) = 40

$ZORDER (Cont.)

The following series of commands shows how $ZO works:

```
SET ^A(1,2)=22
WRITE $ZO
30
```

$ZREFERENCE

PURPOSE

The $ZREFERENCE variable yields the full reference of the current global reference.

FORM:

$ZR{EFERENCE}

EXPLANATION:

The DSM-11 operating system maintains the current global reference of the last referenced global. The $ZR variable returns a full reference (name and subscripts) for this global node.

This is a full reference, containing a UCI and a volume set reference if the last global reference was to a global not in the same UCI as the host job.

RELATED:

The $ZORDER Special Variable
Extended Global References (Section 2.4.3)

EXAMPLE:

Global A contains the following nodes and values:

^A(1,1) = 10
^A(1,2) = 20
^A(1,3) = 30
^A(1,4) = 40

The following series of commands shows how $ZR works:

```
SET ^A(1,2)=22
WRITE $ZR
^A(1,2)
```

$ZTRAP

PURPOSE:

$ZTRAP can contain a reference to a line and/or routine to which you want control to pass in the event of an error.

FORM:

$ZT{RAP}

EXPLANATION:

You can use $ZTRAP to trap errors. Set $ZTRAP to an entry reference (a line reference and/or name of a routine) to which you want control to pass if DSM-11 detects an error. You can use name or argument indirection to the line and/or routine you reference.

When an error occurs after you set $ZTRAP, the system transfers control to the line and/or routine referenced in $ZTRAP.

You can use $ZTRAP in a set of nested DOs or XECUTEs. You can set $ZTRAP to different values for each level. When an error occurs in a higher level (nested DO or XECUTE):

1. If $ZTRAP was set for that level, control passes to the line and/or routine referenced by that $ZTRAP.

2. If $ZTRAP was not set for that level, the DSM-11 error handler examines $ZTRAP for each lower level until it finds a $ZTRAP that is set to a line and/or routine reference.

3. If no $ZTRAP is set at any level, then the DSM-11 default error processing goes into effect.

4. The nested contexts of DOs and XECUTEs are retained. If the error is satisfactorily resolved by the programmer's error-handling routine, then control returns to the routine in which $ZTRAP was set; and execution of the routine can continue.

See the *DSM-11 User's Guide* for more information on error handling.

You can examine $ZERROR to determine which error occurred and where it occurred.

$ZTRAP (Cont.)

COMMENTS:

You have three methods of exiting from an error-handling routine declared in the $ZTRAP special variable:

1. Use a GOTO command to transfer control to any point, including to the routine that caused the error.

2. Use a QUIT command to return to the point immediately following the DO argument that called the routine that caused the error.

3. Use a ZQUIT command to request that the DSM-11 error handler resume searching through the stacked DO and XECUTE contexts for a previously declared $ZTRAP.

RELATED:

The $ZERROR Special Variable
The ZQUIT Command

EXAMPLE:

The following example sets $ZTRAP. When an error occurs, DSM-11 passes control to the second line after the line labeled ERR in the routine CHECK.

```
J1    S $ZT="ERR+2^CHECK"
```

$ZVERSION

PURPOSE:

$ZVERSION contains the name and version of your DSM system.

FORM:

$ZV{ERSION}

EXPLANATION:

$ZVERSION contains a string that identifies the version of the DSM system you are currently using. The string is in the form:

SYSTEM(SP)*Version*(SP)*number*

COMMENTS:

You can use $ZV as a test for the current system in routines to convert applications from one version of DSM to another.

RELATED:

None.

EXAMPLES:

The following example writes the contents of $ZVERSION on the current output device.

```
W $ZV
DSM-11 Version 3.0
```

Appendix A
ASCII Character Set

This appendix lists the characters of the ASCII character set and their octal and decimal codes.

ASCII CHARACTER SET

Octal	Decimal	Character
0	0	NUL
1	1	SOH (Backspace) +
2	2	STX (Forward Space) +
3	3	ETX (CTRL/C)* (Write Tape Mark) +
4	4	EOT (Write Block) +
5	5	ENQ (Rewind) +
6	6	ACK (Read Block) +
7	7	BELL (Read Label) +
10	8	BS*(Write Header Label)*
11	9	HT (Write EOF Label) +
12	10	LF
13	11	VT
14	12	FF
15	13	CR
16	14	SO
17	15	SI (CTRL/O)*
20	16	DLE
21	17	DC1
22	18	DC2
23	19	DC3

ASCII CHARACTER SET

Octal	Decimal	Character
24	20	DC4
25	21	NAK (CTRL/U)*
26	22	SYN
27	23	ETB
30	24	CAN
31	25	EM
32	26	SUB
33	27	ESC (ALTMODE)*
34	28	FS
35	29	GS
36	30	RS
37	31	US
40	32	Space (blank)
41	33	!
42	34	"
43	35	#
44	36	$
45	37	%
46	38	&
47	39	'
50	40	(
51	41	)
52	42	*
53	43	+
54	44	,
55	45	-
56	46	.
57	47	/
60	48	0
61	49	1
62	50	2
63	51	3
64	52	4
65	53	5
66	54	6
67	55	7
70	56	8
71	57	9
72	58	:
73	59	;
74	60	<
75	61	=
76	62	>
77	63	?
100	64	@
101	65	A
102	66	B

ASCII CHARACTER SET

Octal	Decimal	Character
103	67	C
104	68	D
105	69	E
106	70	F
107	71	G
110	72	H
111	73	I
112	74	J
113	75	K
114	76	L
115	77	M
116	78	N
117	79	O
120	80	P
121	81	Q
122	82	R
123	83	S
124	84	T
125	85	U
126	86	V
127	87	W
130	88	X
131	89	Y
132	90	Z
133	91	[
34	92	\
35	93	]
36	94	^
37	95	_
40	96	`
41	97	a
42	98	b
43	99	c
44	100	d
45	101	e
46	102	f
47	103	g
50	104	h
51	105	i
52	106	j
53	107	k
54	108	l
55	109	m
56	110	n
57	111	o
60	112	p
61	113	q

ASCII CHARACTER SET

Octal	Decimal	Character
62	114	r
63	115	s
64	116	t
65	117	u
66	118	v
67	119	w
70	120	x
71	121	y
72	122	z
73	123	{
74	124	|
75	125	}(ALTMODE)*
76	126	*(ALTMODE)*
77	127	DEL (RUBOUT)+

+ denotes control function for magnetic-tape devices

* denotes control function for DSM-11 terminals, if different from specified or other use

Appendix B
DSM-11 Language Summary

This appendix presents a summary of all DSM-11 language elements.

B.1 DSM-11 Syntax Summary

- Every command line starts with a command:

  >*command*

- Every routine line starts with either a label (optional), a TAB, or a CTRL/I, there can be optional spaces between the tab and the command:

  >{*label*}TAB{SP...SP}*command*

- Routine lines in indented execution blocks for block-structured programming contain one or more periods after the TAB:

  >{*label*}TAB PERIOD{PERIOD...PERIOD} *command*

- A command determines the interpretation of its associated syntax.

- You can abbreviate an ANSI Standard command to its first letter:

  C{OMMAND}

- You can abbreviate an extended command to its first two letters:

  ZC{OMMAND}

- Separate an argumentless command from any commands that follow it with two spaces:

  command SP SP*command*

- Separate a command that takes an argument, from its argument or argument list with a single space:

  command (SP)*argument list*

- Separate multiple arguments in a command from each other with commas and no spaces:

  command (SP)*argument 1,argument 2,argument 3*

- For all commands allowing multiple arguments, the form:

  command (SP)*argument 1,argument 2*

  is equivalent in execution to:

  command (SP)*argument 1* (SP)*command*(SP)*argument 2*

  where *command* is the same in both instances.

- Separate multiple commands on a line from each other with one or more spaces:

  command 1 (SP)*argument* (SP){(SP)...(SP)}*command 2* (SP)*argument*

- To make the execution of all commands (except ELSE, FOR, and IF) conditional, append a postconditional expression to the command name. Separate the command name and the postconditional expression with a colon:

  command:postcond (SP)*argument*

- To make the execution of an argument of the DO, GOTO, or XECUTE commands conditional, append a postconditional expression to the argument. Separate the argument and the postconditional expression with a colon:

  ...argument:postcond

- You can abbreviate an ANSI Standard function to its first two characters:

  $F{UNCTION}

- You can abbreviate an extended function to its first three characters:

  $ZF{UNCTION}

- Enclose all function arguments in parentheses and place them immediately after the function name:

  $function(argument list)

- Separate multiple function arguments from each other by commas. Do not include spaces between arguments:

  $function(argument1,argument2,argument3)

- Abbreviate an ANSI Standard special variable to its first two characters:

$V{ARIABLE}

- Abbreviate an extended special variable to its first three characters:

  $ZV{ARIABLE}

- DSM-11 evaluates and executes all command lines in a left-to-right order, except where explicitly stated and where command execution directs otherwise.

- You can append comments to any line, provided the comments are prefixed by a semicolon. Comments after an argument list can have spaces preceding the semicolon:

  ..command SP *argument* {SP...SP};*comment*

- If there are no comments at the end of a line, there should not be any trailing spaces after the last command:

  ..command SP *argument*

- If there are no commands in a routine line that has a comment, the comment must be separated from the label or the beginning of the line by a TAB; there can be optional spaces after the tab and before the comment:

  {*label*}TAB{SP...SP};*comment*

- Argumentless commands require two spaces before the semicolon that precedes the comment:

  command SP SP;*comment*

- Initial command letters that are unused are reserved for future enhancements of the DSM-11 language.

B.2 DSM-11 ANSI Standard Commands

Command	Description
B{REAK}{:*postcond*}	Suspends execution of a routine.
B{REAK}{:*postcond*}SP*argument*	Controls the suspension of execution from a terminal, and enabling of Version 2 of DSM-11 error processing.

where *argument* can be:

expression
@expr atom

C{LOSE}{:*postcond*}SP*argument,...*	Releases ownership of a specified device.

where *argument* can be:

device{:*params*}
@expr atom

D{O}{:*postcond*}

Transfers control to an indented execution block.

D{O}{:*postcond*}SP*argument,...*

Transfers control to a specified line and/or routine. (Returns control to point of origin.)

where *argument* can be:

entry ref{:*postcond*}
@expr atom

E{LSE}

Conditionally executes the statements following it depending on the truth value in $TEST.

F{OR}SP*local*=*parameter,...*

Controls the repeated execution of all commands following it on the line.

where *parameter* can be:

expression
start:step
start:step:stop

G{OTO}{:*postcond*}SP*argument,...*

Transfers control to a specified line or routine. (Control does not return to point of origin.)

where *argument* can be:

entry ref{:*postcond*}
@expr atom

H{ALT}{:*postcond*}

Terminates use of DSM-11. (Unlocks all variables and closes all devices.)

H{ANG}{:*postcond*}SP*argument,...*

Suspends execution for a specified number of seconds.

where *argument* can be:

seconds
@expr atom

I{F}	Permits the conditional execution of the statement or statements that follow it depending on truth value in $TEST.
I{F}␣*argument,...*	Permits the conditional execution of the statement or statements that follow it depending on the truth value of its argument(s).

where *argument* can be:

truth value
@expr atom

J{OB}{:*postcond*}␣*argument,...*	Starts a specified routine in a new partition.

where *argument* can be:

entry ref{[*UCI*{,*volume set*}]}{:*size*}{:*timeout*}
@expr atom

K{ILL}{:*postcond*}	Deletes all local variables.
K{ILL}{:*postcond*}␣*name*	Deletes the local and global variables named as arguments.
K{ILL}{:*postcond*}␣(*local,...*)	Deletes all local variables but those named as arguments.
L{OCK}{:*postcond*}	Unlocks all locked variables.
L{OCK}{:*postcond*}␣*argument,...*	Locks a specified variable or portion of a specified variable.

where *argument* can be:

name{:*timeout*}
(*name,...*){:*timeout*}
@expr atom

N{EW}{:*postcond*}	Saves all local variables, and restores them when the next QUIT is reached.
N{EW}{:*postcond*}␣*name,...*	Saves the local variables named as arguments, and restores them when the next QUIT is reached.

N{EW}{:*postcond*}␣*(local,...)*

Saves all local variables but those named as arguments, and restores the saved local variables when the next QUIT is reached.

O{PEN}{:*postcond*}␣*argument,...*

Obtains ownership of specified devices.

where *argument* can be:

device
device:{*params*}:*timeout*
@expr atom

Q{UIT}{:*postcond*}

Terminates the current flow of execution.

R{EAD}{:*postcond*}␣*argument,...*

Reads input data from the current device.

where *argument* can be:

format
string
local{:*timeout*}
local{#*inn field*}{:*timeout*}
**local*{:*timeout*}
@expr atom

S{ET}{:*postcond*}␣*argument,...*

Assigns the value of an expression to a variable, or to a substring within a variable.

where *argument* can be:

storage ref = *expression*
(storage ref,...) = *expression*
piece ref = *expression*
@expr atom

U{SE}{:*postcond*}␣*argument*

Makes the specified device the current I/O device.

where *argument* can be:

device{:*params*}
@expr atom

V{IEW}{:*postcond*}␣*argument,...*

Reads and writes to memory or disk.

where *argument* can be:

{-}block addr:disk
{-}block addr:volume set
{-}block addr
destination:{state}:value
destination:{state}:source:{state}:extent
@expr atom

W{RITE}{*:postcond*}␣*argument,...* — Writes data and/or control information to the current I/O device.

where *argument* can be:

form
expression
**integer*
@expr atom

X{ECUTE}{*:postcond*}␣*argument,...* — Executes the one-line subroutine specified in the argument.

where *argument* can be:

expression{*:postcond*}
@expr atom

B.3 Extended DSM-11 Commands

ZA{LLOCATE}{*:postcond*}␣*argument: {timeout}...*
Makes specified variables or specified nodes of variables unavailable for allocation (locking) by other users.

where *argument* can be:

name
expr atom

ZB{REAK}{*:postcond*}␣*control element*
Turns on, turns off, and controls the *DSM-11 debugger.*

ZDEALLOCATE{*:postcond*}

ZDEALLOCATE{*:postcond*}␣{*argument*}

	Unlocks the specified variables; these variables were previously locked with the ZALLOCATE command.

where *argument* can be:

name
@expr atom

ZG{O}{:*postcond*}	Resumes execution of a routine after a BREAK command.
ZG{O}{:*postcond*}SP{*entry ref*}	Starts a debugging session at a specified entry reference.
ZI{NSERT}{:*postcond*}SP*argument,...*	Inserts a new line into the routine in memory.

where *argument* can be:

string{:*line ref*}
string{:*line spec*}
@expr atom

ZL{OAD}{:*postcond*}	Loads a routine from the current device into memory.
ZL{OAD}{:*postcond*}SP*argument,...*	Loads a routine in the routine directory into memory.

where *argument* can be:

routine
@expr atom

ZP{RINT}{:*postcond*}	Writes the current routine on the current device.
ZP{RINT}{:*postcond*}SP*argument,...*	Writes specified lines from the current routine on the current device.

where *argument* can be:

start ref{:*end ref*}
start spec{:*end spec*}
@expr atom

ZQ{UIT}{:*postcond*}	Directs control to the previously declared error-handling routine.

ZR{EMOVE}{:*postcond*}	Deletes the current routine.
ZR{EMOVE}{:*postcond*}SP*argument,...*	Deletes the specified line or lines in the current routine.

where *argument* can be:

start ref{:*end ref*}
start spec{:*end spec*}
@expr atom

ZS{AVE}{:*postcond*}	Stores the current, already named routine.
ZS{AVE}{:*postcond*}SP*argument*	Stores the current routine under the name specified in the argument.

where *argument* can be:

routine
@expr atom

ZT{RAP}SP{*argument*}	Forces an error when it is encountered by the interpreter.
where *argument* is:	an informational message to be included in the error message string.
ZU{SE}{:*postcond*}SP*argument*	Allows you to write to a terminal owned by another user.

where *argument* can be:

terminal
@expr atom

ZW{RITE}{:*postcond*}	Writes the currently defined local variables to the current device.
ZW{RITE}{:*postcond*}SP*argument*	Wr ites the specified local variables to the current device.

where *argument* can be:

local
@expr atom

B.4 DSM-11 Functions

Function	Description
$A{SCII}(*string*{,*position*})	Returns the code of the specified ASCII character as a decimal integer.
$C{HAR}(*ASCII code,...*)	Returns the ASCII equivalent of the specified integer(s).
$D{ATA}(*storage ref*)	Returns a nonnegative integer value indicating whether or not a variable has a defined value and/or descendants.
$E{XTRACT}(*string,*{*start pos*{,*end pos*}})	Returns a specified portion of a specified string.
$F{IND}(*string,substring*{,*position* })	Returns an integer number that is the character position immediately to the right of the rightmost character of the specified substring.
$J{USTIFY}(*string,field*{,*dec field*})	Returns a string right-justified in a field of n spaces.
$L{ENGTH}(*string*{,*delimiter*})	Returns an integer value equal to the number of characters in the specified string or to the number of substrings in the specified string.
$N{EXT}(*storage ref*)	Returns the next subscript at the same subscript level as the specified array node. (Returns subscripts of local variables in numeric collating sequence.)
$ORDER(*storage ref*)	Returns the next subscript at the same subscript level as the specified array node.
$P{IECE}(*string,delimiter* {,*range*})	Returns one or more fields of a string.

where *range* can be:

start field{,*end field*}

$R{ANDOM}(*integer*)	Returns a pseudorandom integer whose value is between 0 and integer-1.
$S{ELECT}(*truth value :expression,...)*	Returns the value of the first expression whose associated truth-value is true.
$T{EXT}(*argument*)	Returns a string that contents of the routine line specified in the $TEXT argument.

where *argument* can be:

line ref
line spec

$V{IEW}(*location*{,*state*})	Returns an integer that is the decimal value of the contents of a particular memory location.
$ZC{ALL}(*external ref*{,*expression,...*}*)*	Provides an interface from DSM-11 to user-written functions.
$ZN{EXT}(*global ref*)	Performs a physical scan of an array.
$ZO{RDER}(*global ref*)	Performs a physical scan of an array.
$ZS{ORT}(*storage ref*)	Returns the subscript of the next sibling node to the node specified. (Returns local variables in string collating sequence.)
$ZU{CI}(*name*{,*volume set name*})	Returns the UCI number, given the name of a UCI.
$ZU{CI}(*number*{,*volume set number*})	the UCI name, given the number of a UCI.

B.5 DSM-11 Special Variables

Variable	Description
$H{OROLOG}	Contains the date and time.
$I{O}	Contains the current I/O device.
$J{OB}	Contains the job number.
$S{TORAGE}	Contains the number of bytes of free space remaining in the partition.
$T{EST}	Contains the truth-value of the most recently executed IF command with an argument, the truth-value from an OPEN, LOCK or READ command with a timeout, or JOB command.
$X	Contains a nonnegative integer value that is equivalent to the value of a carriage or horizontal cursor position on the current device.
$Y	Contains a nonnegative integer value equivalent to the current line on the current page of the current device.
$ZA	Contains a value whose meaning is dependent upon the identity of the current I/O device.
$ZB	Contains a value whose meaning is dependent upon the identity of the current I/O device.
$ZB{REAK}{(*n*)}	Contains a reference to a line, a routine, and a command that DSM-11 interprets as a breakpoint when the debugger is activated.
where *n* is:	A number from 0 to 9

$ZB{REAK}(0)	Contains an action string that is executed after each single step in Break Mode.
$ZE{RROR}	Contains the error code, line reference, routine name, and command number of the most recent error.
$ZO{RDER}	Contains the value of the next global node in sequence after the current global reference.
$ZR{EFERENCE}	Contains the full reference of the current global reference.
$ZT{RAP}	Can be set to a routine reference for error trapping and processing.
$ZV{ERSION}	Contains the name and version of the current system.

B.6 DSM-11 Operators

Operator	Symbol	Example	Meaning
Binary Add	+	A+B	Add B to A.
Binary And	&	A&B	Test if A and B are true.
Binary Concatenate	_	A_B	Concatenate A and B.
Binary Contains	[	A[B	Test if A contains B.
Binary Divide	/	A/B	Divide A by B.
Binary Equals	=	A=B	Test if A and B are equal.
Binary Follows	]	A]B	Test if A follows B.
Binary Greater Than	>	A>B	Test if A is greater than B.

Binary Inclusive Or	!	A!B	Test if either A or B are true.
Binary Integer Divide	\	A\B	Integer divide A by B.
Binary Less Than	<	A<B	Test if A is less than B.
Binary Modulo	#	A#B	A modulo B.
Binary Multiply	*	A*B	Multiply A by B.
Binary Pattern Match	?	A?patn	Test if A fits pattern specified by patn to the right of the ?.
Binary Subtract	-	A-B	Subtract B from A.
Indirection	@	@B	Evaluate the value of B.
Partial Indirection	@	@B@ (subscript)	Evaluate the value of B, do not evaluate subscript.
Unary Minus	-	-B	B is a numeric value of the opposite sign.
Unary Not	'	'B	B is logically inverted.
Unary Plus	+	+B	B is a numeric value.

B.7 DSM-11 Special Symbols

Symbol	Definition
#	A multiply-defined character; can be: 1. A formatting character that causes a form feed on the current device. 2. The Binary MODULO operator.
!	A multiply-defined character; can be: 1. A formatting character that causes a new line operation on the current device. 2. The Binary INCLUSIVE OR operator.

?

A multiply-defined character; can be:

1. A formatting character that is followed by a numeric expression indicating the number of spaces to tabulate in from the left margin.
2. The Binary PATTERN MATCH operator.

, (comma)

The separator in an argument list or a subscript list.

. (period)

An indicator of an indented execution block in block-structured programming.

SP

A multiply-defined separator:

1. One space separates a command from its arguments.
2. One or more spaces separate the last argument of a preceding command from the next command on the line.
3. Two or more spaces separate an argumentless command from the next command on the line.

A multiply-defined character; can be:

1. A delimiter for the various parts of a FOR parameter.
2. An indicator of the presence of an optional postconditional expression appended to a command or argument of a command.
3. An indicator of the presence of a timeout appended to the argument of a command.
4. An indicator of optional parameters for the OPEN, CLOSE, and USE commands.
5. An indicator of a range of lines in a ZPRINT and ZREMOVE argument.
6. A separator of the two portions of a ZINSERT argument.
7. A separator of the two portions of a $SELECT argument.

;

A delimiter indicating that the remainder of the line is a comment.

$

A multiply-defined character; can be:

1. The first character of a function name.

2. The first character of a system-variable name.

" A delimiter of literal strings.

^ Precedes a global reference or the name of a routine not in memory.

= A multiply-defined character; can be:

1. The Binary EQUALS Operator.
2. An assignment operator with the SET command that gives the operand on the left the value of the operand on the right.

Glossary

Ancestor

Any node in an array on a higher level than a given node and on a direct path between that node and the root of the array. For example, the nodes A(1), A(1,2), and A(1,2,5) are ancestors of the node A(1,2,5,6).

ANSI

An acronym for the American National Standards Institute.

Arithmetic Operator

An operator that gives its associated operand or operands a numeric interpretation. Arithmetic operators produce a result that is the value produced by the specified arithmetic operation on the values of their operands.

Argument

An expression that controls the action, location, direction, or range of the command or function with which it is used.

Array

An ordered set of local or global elements or nodes referenced by subscripts and a common variable name.

ASCII

The American Standard Code for Information Interchange. A series of 128 binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol. The ASCII set is defined in ANSI Standard X3.4-1968.

ASCII Graphic Characters

The characters in the ASCII set that the system can write as the characters they represent. The graphic characters are ASCII decimal values 32 through 126.

Canonic Number

A number reduced to its simplest form. A canonic number contains only valid numeric characters and, optionally, a single decimal point. It has no leading zeros to the left of the integer or trailing zeros to the right of the fraction. A canonic integer has no decimal point.

Collating Sequence

An ordering assigned to a set of items. For arrays, DSM-11 can collate nodes in a numeric or in an ASCII sequence.

In numeric sequence, DSM-11 first orders all nodes with subscripts that are canonic numbers in the ascending numeric order of their subscripts evaluated as numbers. It then orders all nodes with subscripts that are strings or noncanonic numbers (such as 01) in the ascending order of their ASCII-character values.

In ASCII collating sequence, DSM-11 orders all nodes in the ascending order of the subscript's ASCII-character values.

Column

The vertical position on a terminal screen (X orientation from left to right) at which a character is displayed.

Command

A name or mnemonic that, by virtue of its syntax and position on an input line, causes the DSM-11 interpreter to perform a predefined action.

Command Line

One or more statements input to DSM-11 for immediate execution. DSM-11 distinguishes a command line from a routine line (entered for later execution) by the absence of a line label and TAB character.

Comment

A brief, nonexecuted explanation of the purpose of a routine or of a routine line. You must precede all comments with a semicolon. You can use any valid DSM-11 graphic character in a comment. You cannot use control characters in a comment.

Concatenation

The process of evaluating two or more string expressions to produce a single string that is a joining of the values of the expressions. Concatenation is designated by the underline character.

Control Characters

The result of pressing the CTRL key and a letter key simultaneously.

Constant

Another term for numeric literal.

Current Device

The value of the $IO special variable. All READ, WRITE, ZWRITE, ZPRINT, and argumentless ZLOAD command occur through the current device. The value of $IO is set as follows:

1. If no USE command has ever been issued, $IO is set to the principal device.
2. The USE and ZUSE commands set $IO.
3. A CLOSE command with an argument equal to $IO causes $IO to be set to the principal device.

Cursor

A unique symbol, usually a flashing underscore, that marks your current position on a video terminal. Whatever character you type next appears at the current cursor position.

Data

The information (facts, numbers, and symbols) entered into the system for processing and/or storage. DSM-11 has only one data type: the variable length string. (DSM-11 regards numbers as a special interpretation of a string.)

Descendant

For any array node (element), any other array node on a lower (deeper) subscripting level that can be reached from that node and that shares the first n subscripts in common with that node. For example, the nodes A(1,2,2) and A(1,2) are descendants of A(1).

Device

Any part of the computer system other than the central processing unit, memory, or their associated structures. A device can be a physical piece of equipment, such as a terminal or a line printer, or a logical piece of equipment, such as the Sequential Disk Processor or In-Memory Job Communication.

DSM-11

The acronym for DIGITAL Standard MUMPS, Digital Equipment Corporation's implementation of ANSI Standard MUMPS for the PDP-11.

Entry Reference

A reference to a line and/or routine used as an argument to the DO or GOTO command. The entry reference can be:

- A line label
- A line label and offset
- A routine name
- A line label and routine name
- A line label, offset, and routine name

Evaluation Order

The way in which DSM-11 evaluates expressions. Normally, DSM-11 evaluates all expressions in the following sequence:

1. All occurrences of indirection
2. All unary operators
3. All binary operators

All binary operators have the same priority. You can use parentheses to modify the normal left-to-right evaluation order of binary operators.

Exponential Notation

A shorthand method of expressing very large or very small numeric quantities similar to scientific notation. The format for exponential notation is:

number E*exponent*

DSM-11 can interpret numeric literals or numeric data entered in exponential notation format; however, it does not produce the results of mathematical calculations in exponential notation format.

Expression

A string of characters that yields a value when executed. An expression contains at least one expression atom optionally followed by other expression atoms separated by binary operators.

Expression Atom

A unit used to build expressions. An expression atom can consist of a single variable, literal, function, an expression atom preceded by a unary operator, or an expression in parentheses.

Function

A name preceded by a dollar sign ($) and followed by an argument or argument list enclosed in parentheses. The function and its argument list define and perform a procedure.

Global

Another name for a global variable.

Global Variable

A named reference to data on storage media. A global variable can be either simple (a variable name that references a single datum) or subscripted (a variable name that references an array). Global variables are not unique to a user. They can be examined or changed by any authorized user.

Indirection

A means of using the value of an expression in DSM-11 routines. Indirection is denoted by the at (@) sign followed by an expression atom.

Whenever DSM-11 finds an occurrence of indirection, it substitutes the value previously given the expression atom for the occurrence of indirection and evaluates the value in the context of the occurrence. This allows DSM-11 to execute the same segment of code repeatedly for different values of the expression atom.

Job

Any system activity that requires the use of a partition; for example, logging into the system or issuing a JOB command.

Line

A string of characters terminated with a valid line terminator.

Line Label

An optional name at the beginning of a routine line that identifies the line within that routine. A line label should be unique to a routine and should have no more than eight characters.

If the first character of the line label is an uppercase alphabetic character or a percent (%) character, the remaining characters in the label can be any combination of uppercase alphabetics or digits. If the first character of the line label is a digit, the remaining characters in the label must be digits.

Line Reference

A reference to a line within a DSM-11 routine. A line reference can be:

- A line label
- A line label and offset

Literal

A string of characters occurring within the context of a routine that never changes value from one execution of the routine to another. DSM-11 recognizes two types of literals:

- Numeric literals
- String literals

Local Variable

A variable that exists only in memory. Local variables are unique to a user and can usually be inspected or changed only by that user.

Logical Operator

An operator that produces a truth-valued result based on the truth value of its operand or operands.

First, DSM-11 interprets each operand numerically. If the operand evaluates to any value but zero, DSM-11 gives it a value of true (one). If an operand evaluates to zero, DSM-11 gives it a value of false (zero).

Finally, DSM-11 evaluates the truth values of the operands in terms of the relationship described by the operator. If the relationship is true, DSM-11 produces a result of true (one). If the relationship is false, DSM-11 produces a result of false (zero).

MUMPS

An acronym for Massachusetts General Hospital Utility Multi-Programming System.

Naked Indicator

The value of a previous global reference returned so that full references can be constructed from naked references. The naked indicator records the last global reference and its level within the array.

Naked Reference

A shorthand method of referring to a node in a global array. Naked references permit you to refer to the sibling or descendant of a previously referenced node by using only the circumflex (^) character and the unique portion of the sibling's or descendant's subscript.

Name

A term that applies to certain elements in DSM-11. A name should consist of no more than eight uppercase alphabetic or digit characters. The first character in a name must be either an uppercase alphabetic or a percent (%) character.

DSM-11 recognizes the following elements as names:

- Routine names
- Line labels
- Variable names
- External references

Node

A global or local array element addressed by the name (common to all members of the array) and a unique subscript.

Numeric Expression

An expression evaluated numerically. DSM-11 evaluates an expression numerically when it contains a binary or unary arithmetic operator.

Numeric Literal

A literal evaluated mathematically as an integer or decimal number. A numeric literal is one that contains the digits 0 through 9, the decimal point, the Unary MINUS operator, the Unary PLUS operator, or the letter E. DSM-11 accepts numeric literals with the range plus or minus 10 to the power of plus or minus 26.

Operand

An expression or expression atom used by an operator to produce a result.

Operator

A symbolic character that indicates the action to be performed and the type of value to be produced from the value(s) of its associated operand(s).

Parent

For any node in an array, the ancestor on the next higher level of an array. For example, the node A(1,2,3) is the parent of the nodes A(1,2,3,1), A(1,2,3,3) and A(1,2,3,4).

Partition

The area of memory that contains all elements of a job. Partitions contain routine and local-variable storage areas and status information about routines.

Postconditional Expression

A truth-valued expression appended to a command or argument that makes the execution of that command or argument dependent on the truth value of the expression. If the postconditional expression is false, DSM-11 does not execute the command or argument. If the postconditional expression is true, DSM-11 does execute the command or argument.

Principal Device

The device on which the system conducts all I/O operations in the absence of specific OPEN and USE or ZUSE commands to the contrary. For interactive users, the principal device is the terminal on which they logged into the system.

Prompt

A word or message printed by the system that requests some action on your part.

Relational Operator

An operator that produces a truth-valued result based on the relationship of its operands. If the relationship expressed by the operator and its operands is

true, it produces a value of true (one). If the relationship expressed by the operator and its operands is false, it produces a value of false (zero).

Root

The first, unsubscripted name level of an array.

Routine

A collection of DSM-11 lines that are saved, loaded, called (using DO), or overlaid (using GOTO) as a single unit. A set of computer instructions or symbolic statements combined to perform a task.

Routine Line

One or more statements input to DSM-11 for later execution as part of a routine. You must precede the statements in a routine line with a TAB character. You can precede the TAB with an optional line label.

Sibling

For any node in an array, all nodes that have the same immediate ancestor (parent). Siblings have the same number of subscripts and differ only in the last subscript. For example, the nodes B(2,3,4), B(2,3,6), and B(2,3,7) are siblings because they share the same parent, B(2,3).

Sparse Array

An array that need not be predefined to its maximum size. A sparse array contains only those nodes that have been explicitly defined.

The sparse array is a dynamic structure. As more nodes are defined, DSM-11 allocates space for them. As nodes are deleted, DSM-11 deallocates space for them.

Special Variable

A variable that is permanently defined within DSM-11. These variables provide system and control information. The first character of a special variable is always a dollar sign ($).

Statement

A unit consisting of a command (and --- optionally --- its modifying arguments) that specifies an operation to be performed.

String

A contiguous set of up to 255 ASCII characters. DSM-11 stores all data as variable-length strings.

String Data

Any set of between 0 and 255 characters taken as a data entry and referenced by a local or global variable.

String Expression

An expression which DSM-11 does not give a special (numeric or logical) interpretation.

String Literal

A string of characters enclosed in double quotation marks within the context of a line. The value of a string literal is a function of its spelling.

Storage Reference

A name of a storage location that has the value of the contents of that location. DSM-11 recognizes four types of storage reference:

- Local variables
- Global variables
- Naked references
- Special variables

Subroutine

A group of statements arranged so that control can pass to the subroutine and back to the main routine again. Subroutines usually perform tasks required more than once in a routine.

Substring

Any contiguous part of a specified string.

Subscript

A numeric or string value appended to a local or global variable name to identify a specific element or node in an array. Subscripts are enclosed in parentheses. Multiple subscripts must be separated by commas.

Subscripted Variable

A local or global variable to which a subscript is affixed. An element or node in a local or global array.

Tree Structure

The structure of DSM-11 arrays. The tree structure is drawn like a family tree with the root (name) at the top and the nodes arranged below by their depth of subscripting. All nodes with one subscript are on the first level, all nodes with two subscripts are on the second level, and so forth.

The tree structure is only a logical picture of an array; it does not reflect how DSM-11 physically stores the array. For example, the tree structure shows all pointer nodes and defined data nodes. DSM-11 does not store such pointer nodes, but does recognize their logical existence.

Truth-Valued Expression

An expression whose result is a truth-value. A false expression produces a result of zero. A true expression produces a result of one.

Timeout

An integer-valued expression preceded by a colon that can be appended to the argument of an OPEN, LOCK, READ, or JOB command. The integer specifies the number of seconds DSM-11 is to try to complete the operation the command specifies before resuming execution.

Variable

A symbolic name for a location where a datum is stored. DSM-11 recognizes three types of variables:

- Local variables
- Global variables
- Special variables

Local variables are stored in memory. Global variables are stored on media. Special variables are generated by DSM-11 when requested.

INDEX

ADD
  see Binary ADD
AND
  see Binary AND
Arguments
  argument lists, 1-6
  functions, 2-3
  indirection, 2-26
  postconditional expressions, 2-36
  spacing, 1-5
  syntax, 1-5
Arithmetic operators, 2-21
  table of, 2-22
Arrays
  ancestors, 2-14
  collating in ASCII sequence, 2-16
  collating in numeric sequence, 2-16
  collating sequences, 2-15
  $DATA, 6-8
  descendants, 2-14
  functions
    see $NEXT
    see $ORDER
    see $ZNEXT
    see $ZORDER
    see $ZSORT
    comparison of, 6-14
  node states, 2-15, 6-8
  parent, 2-14
  siblings, 2-14
  sparse, 2-13
  structure, 2-13 to 2-17
  tree structure, 2-13
$ASCII, 6-3 to 6-4
ASCII character set, A-1

Binary ADD, 3-2 to 3-3
Binary AND, 3-4 to 3-5
  NAND, 3-4
  NOT AND, 3-4
Binary CONCATENATE, 3-6
Binary CONTAINS, 3-7 to 3-8
Binary DIVIDE, 3-9 to 3-10
Binary EQUALS, 3-11 to 3-13
Binary FOLLOWS, 3-14 to 3-16
Binary GREATER THAN, 3-17 to 3-18
Binary INCLUSIVE OR, 3-19 to 3-20
Binary INTEGER DIVIDE, 3-21 to 3-22
Binary LESS THAN, 3-23 to 3-24
Binary MODULO, 3-25 to 3-27
Binary MULTIPLY, 3-28 to 3-29
Binary operators
  and expression atoms, 2-2, 2-20
  arithmetic operators, 2-21
  Boolean, 2-22
  definition, 2-2
  definition of, 2-20
  expression evaluation, 2-32
    nested parentheses, 2-33
    parentheses, 2-33
  logical operators, 2-24
  numeric relational operators, 2-22
  operands, 2-20
  string operators, 2-24
  string relational operators, 2-23
  type of, 2-21
Binary PATTERN MATCH, 3-30 to 3-33
Binary SUBTRACT, 3-34 to 3-35
Block structuring
  DO, 4-8
  execution levels, 1-14
  execution termination, 1-15
  GOTO, 1-16
  QUIT, 1-15
  syntax, 1-13 to 1-16
Boolean operators, 2-22, 2-23
Boolean values, 2-35

BREAK, 4-3 to 4-5
Break Mode
  $ZBREAK, 7-14
  ZGO, 5-11, 5-12
Breakpoints
  $ZBREAK, 7-13

Canonic number
  definition of, 2-16
$CHAR, 6-5 to 6-7
  cursor control, 6-7
  escape sequences, 6-7
Characters
  ampersand, 2-24
  $ASCII, 6-3
  ASCII character set, 1-2, A-1
  at sign, 2-25, 3-36
  carriage return, 2-29, 2-30
  $CHAR, 6-5
  circumflex and global variables, 2-11
  circumflex in naked references, 2-17
  control B, 5-8, 5-11, 7-14
  control C, 4-3, 4-4, 4-15
  control Y, 4-41
  equals and SET, 2-23
  equals as a relational operators, 2-23
  exclamation mark, 2-24, 2-29, 2-30
  form feed, 2-29, 2-30
  formatting, 2-29
  horizontal tabulation, 2-29, 2-30
  in local variable name, 2-9
  line feed, 2-29, 2-30
  nonprinting, 1-3
  number sign, 2-29, 2-30
  operator summary, B-13
  percent, 2-9
  percent and global variables, 2-11
  period in PATTERN MATCH, 3-31
  question mark, 2-29, 2-30
  single quote, 2-20
  special symbol summary, B-14
  TAB, 1-9
  TAB in routine lines, 1-8
  table of formatting, 2-29
  use of lowercase, 1-2 to 1-3
  use of mixed case, 1-2 to 1-3
  use of uppercase, 1-2 to 1-3
CLOSE, 4-6 to 4-7
Command lines
  syntax, 1-7
Commands
  abbreviations, 1-4
  description of extended, 5-2 to 5-36
  descriptions of ANSI standard, 4-2
  postconditional expressions, 2-35
  summary of ANSI standard, B-3 to B-7
  summary of extended, B-7 to B-9
    syntax, 1-4
Comments
  syntax, 1-9
CONCATENATE
  see Binary CONCATENATE
Conditional statements
  see ELSE
  see IF
CONTAINS
  see Binary CONTAINS
Control B, 7-14
  ZBREAK, 5-8
  ZGO, 5-11
Control C, 4-3, 4-4, 4-15
  recognition
    see BREAK
Control Y
  NEW, 4-41
Cursor control
  with $CHAR, 6-7
Cursor position
  current horizontal position in
    $X, 7-9
  current vertical position in $Y, 7-10

$DATA, 6-8 to 6-9
Data storage
  variable-length strings, 2-8
Date
  $HOROLOG, 7-3
Debugger
  breakpoints, 7-13
  MUMPS, 4-3, 5-7, 5-8, 5-11,
    5-12, 7-13, 7-15
Debugging
  see BREAK
  see ZBREAK
  see ZGO
Disk drive types, 4-66
DIVIDE
  see Binary INTEGER DIVIDE
DO, 4-8 to 4-11

argumentless, 1-13
block-structured programming, 4-8

Editing
ZINSERT, 5-13 to 5-17
ZREMOVE, 5-26 to 5-28
ELSE, 4-12 to 4-13
EQUALS
see Binary EQUALS, 3-11
Error processing, 4-3
$ZA, 7-11
$ZERROR, 7-17
ZQUIT, 5-23
$ZTRAP, 7-21
ZTRAP, 5-31 to 5-32
Escape sequences
with $CHAR, 6-7
Execution levels
block structuring, 1-14
Exponential notation
syntax, 2-6
Expression atoms
definition, 2-1
Expression evaluation
argument postconditional expressions, 2-36
binary operators, 2-32
nested parentheses, 2-33
parentheses, 2-32
command postconditional expressions, 2-36
indirection, 2-32
naked references and postconditional expressions, 2-37
numeric expressions, 2-35
sequence, 2-32
string expressions, 2-34
timeouts, 2-38
truth-valued expressions, 2-35
unary operators, 2-32
Expressions
definition, 2-1
numeric, 2-34
postconditional, 2-35
string, 2-34
truth-valued, 2-35
types of, 2-34
Extended global references
syntax, 2-12
volume sets, 2-12
$EXTRACT, 6-10 to 6-12

$FIND, 6-13 to 6-15
FOLLOWS
see Binary FOLLOWS
FOR, 4-14 to 4-18
Formatting characters
carriage return, 2-29, 2-30
definition of, 2-29
form feed, 2-29, 2-30
horizontal tabulation, 2-29, 2-30
line feed, 2-29, 2-30
table of, 2-29
Free space
$STORAGE, 7-6
Function
definition, 2-2
Functions
abbreviations, 2-3
arguments, 2-3
description of, 6-2 to 6-48
summary, B-10 to B-11
syntax, 2-2

Global variables
definition of, 2-11
limits, 2-12
subscript definition, 2-12
GOTO, 4-19 to 4-21
in block structuring, 1-16
GREATER THAN
see Binary GREATER THAN
GREATER THAN OR EQUAL TO, 3-23

HALT, 4-22
HANG, 4-23 to 4-24
$HOROLOG, 7-3

IF, 4-25 to 4-27
INCLUSIVE OR
see Binary INCLUSIVE OR
Indirection
argument, 2-26, 3-36
definition of, 2-25
expression evaluation, 2-32
INDIRECTION operator, 3-36 to 3-39

name, 2-26, 3-37
nested, 2-28
operator, 2-25
partial, 2-28, 3-38
pattern, 2-27, 3-38
subscript, 2-28, 3-38
syntax of partial, 2-28
types of, 2-26
types of name, 2-26
Input
see READ
$IO, 7-4

$JOB, 7-5
JOB, 4-28 to 4-30
partition size, 4-29
Job number
$JOB, 7-5
$JUSTIFY, 6-16 to 6-18

KILL, 4-31 to 4-33

Labels
line label syntax, 1-8
offsets, 1-11
$LENGTH, 6-19 to 6-20
LESS THAN
see Binary LESS THAN
LESS THAN OR EQUAL TO, 3-17
see Binary GREATER THAN
Line specifications
syntax, 1-12
Lines
comments, 1-9
line labels, 1-8
routine, 1-8
syntax, 1-7
Literals
decimal numeric, 2-4
definition, 2-3
definition of decimal numeric, 2-4
definition of integer, 2-4
definition of numeric, 2-3
integer, 2-4
numeric, 2-3
syntax, 2-3
Local variables
definition of, 2-9
simple, 2-10
subscript definition, 2-10
subscript limits, 2-10
subscripted, 2-10
LOCK, 4-34 to 4-38
Locking variables
see LOCK
see ZALLOCATE
Logical operators
definition of, 2-24
negative logical operator table, 3-42
table of, 2-24
truth table, 2-25
Logout
see HALT
Loops
see FOR

Memory access
see VIEW
Messages
to other users
see ZUSE
MINUS
see Unary MINUS
MODULO
see Binary MODULO
MULTIPLY
see Binary MULTIPLY

Naked references, 2-17
and LOCK, 4-36
and SET, 4-57
circumflex, 2-17
naked indicator, 2-17
postconditional expressions, 2-37
undefined, 2-18
ZALLOCATE, 5-4
Names
indirection, 2-26
NAND, 3-4
NEW, 4-39 to 4-42
$NEXT, 6-21 to 6-22
NOR, 3-19
NOT
see Unary NOT
NOT AND, 3-4
NOT EQUALS, 3-12

NOT LESS THAN, 3-23
NOT OR, 3-19
Numeric expressions
  definition of, 2-34
  evaluation, 2-35
Numeric relational operators
  definition of, 2-22
  table of, 2-22
  Unary NOT, 2-22

Offsets, 1-11
OPEN, 4-43 to 4-45
Operators
  see arithmetic operators
  see binary operators
  see indirection operators
  see logical operators
  see numeric relational operators
  see string relational operators
  see unary operators
  definition of, 3-1
  summary, B-13 to B-14
OR
  see Binary INCLUSIVE OR
$ORDER, 6-23 to 6-25
Output
  see WRITE

Parentheses
  expression evaluation, 2-32
  nested and evaluation, 2-33
Partitions
  free space
    $STORAGE, 7-6
  size with JOB, 4-29
PATTERN MATCH
  see Binary PATTERN MATCH
$PIECE, 6-26 to 6-29
PLUS
  see Unary PLUS
Postconditional expressions
  argument, 2-36
  argument evaluation, 2-36
  argument syntax, 2-36
  command, 2-35
  command evaluation, 2-36
  command syntax, 2-35
  definition of, 2-35
  naked references, 2-37
Program control
  see DO
  see GOTO

QUIT, 4-46 to 4-47
  in block structuring, 1-15

$RANDOM, 6-30
READ, 4-48 to 4-53
References
  entry, 1-11
  entry references syntax, 1-12
  extended global, 2-12
  line, 1-11
  line specifications, 1-11
  naked, 2-17
  syntax, 1-11
Routine calling
  see DO
Routine lines
  comments, 1-9
  syntax, 1-8
Routines
  syntax, 1-10

Saving routines
  see ZSAVE
Saving variables
  see NEW
$SELECT, 6-31 to 6-32
SET, 4-54 to 4-60
Significant digits
  mathematical operations, 2-6
Spacing
  arguments, 1-5
  statements, 1-4
Special symbols
  summary, B-14 to B-16
Special variables
  abbreviations, 2-19
  and SET, 4-57
  definition of, 2-19
  description of, 7-2 to 7-23
  summary, B-12 to B-13
  type of, 2-19
Statements
  spacing, 1-6

syntax, 1-4
$STORAGE, 7-6
String expressions
definition of, 2-34
evaluation, 2-34
String literals
definition of, 2-7
String manipulation
see $EXTRACT
see $FIND
see $JUSTIFY
see $LENGTH
see $PIECE
String operator, 2-24
Binary CONCATENATE, 3-6
String relational operators
definition of, 2-23
table of, 2-23
Unary NOT, 2-23
Subscripts
definition of global variable, 2-12
indirection, 2-28
local variables, 2-10
local variables definition, 2-10
local variables limits, 2-10
SUBTRACT
see Binary SUBTRACT, 3-34
Summary
ANSI standard commands, B-3 to B-7
extended commands, B-7 to B-9
functions, B-10 to B-11
language, B-1 to B-16
operators, B-13 to B-14
special symbols, B-14 to B-16
special variables, B-12 to B-13
syntax, B-1 to B-3
Syntax
argument lists, 1-6
argument postconditional expressions, 2-36
arguments, 1-5
block structuring, 1-13 to 1-16
command lines, 1-7
command postconditional expressions, 2-35
commands, 1-4
comments, 1-9
entry references, 1-12
exponential notation, 2-6
extended global references, 2-12
labels, 1-11
line labels, 1-8
line references, 1-11
line specifications, 1-12
lines, 1-7
literals, 2-3
numeric literals, 2-3
offsets, 1-11
partial indirection, 2-28
references, 1-11
routine lines, 1-8
routines, 1-10
statement spacing, 1-6
statements, 1-4
summary, B-1 to B-3
timeout expressions, 2-37

$TEST, 7-7 to 7-8
$TEXT, 6-33 to 6-34
Time
$HOROLOG, 7-3
Timeout expressions
definition of, 2-37
expression evaluation, 2-38
syntax, 2-37
Tracing errors
see BREAK
Truth-valued expressions
definition of, 2-35
evaluation, 2-35

Unary MINUS, 3-40 to 3-41
Unary NOT, 3-42 to 3-44
Unary operators, 2-20
and expression atoms, 2-20
definition of, 2-20
expression evaluation, 2-32
table of, 2-20
Unary PLUS, 3-45 to 3-46
USE, 4-61 to 4-63

Variable-length strings
data storage, 2-8
Variables
definition of, 2-8
definition of global, 2-11
definition of local, 2-9
global subscript definition,

2-12
global subscript limits, 2-12
global variables, 2-11
local simple, 2-10
local subscript limits, 2-10
local subscripted, 2-10
special, 2-19
types of variables, 2-9
$VIEW, 6-35 to 6-37
VIEW, 4-64 to 4-70
Volume sets
extended global references, 2-12
VT100 cursor control
with $CHAR, 6-7
VT100 escape sequences
with $CHAR, 6-7

WRITE, 4-71 to 4-73
asterisk form, 4-72
star form, 4-72

$X, 7-9
XECUTE, 4-74 to 4-76

$Y, 7-10

$ZA, 7-11
ZALLOCATE, 5-3 to 5-6
$ZB, 7-12
$ZBREAK, 7-13 to 7-16
ZBREAK, 5-7 to 5-8
$ZCALL, 6-38
ZDEALLOCATE, 5-9 to 5-10
$ZERROR, 7-17
ZGO, 5-11 to 5-12
ZINSERT, 5-13 to 5-17
ZLOAD, 5-18 to 5-19
$ZNEXT, 6-39 to 6-40
$ZORDER, 6-41 to 6-44, 7-18 to 7-19
ZPRINT, 5-20 to 5-22
ZQUIT, 5-23 to 5-25
$ZREFERENCE, 7-20
ZREMOVE, 5-26 to 5-28
ZSAVE, 5-29 to 5-30
$ZSORT, 6-45 to 6-46
$ZTRAP, 7-21 to 7-22
ZTRAP, 5-31 to 5-32
$ZUCI, 6-47 to 6-48
ZUSE, 5-33 to 5-34
$ZVERSION, 7-23
ZWRITE, 5-35 to 5-36

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear – Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1–2/L12
MARLBOROUGH, MA 01752

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line