VAX OPS5 Reference Manual

Order Number: AA–EZ19C–TE

May 1989

This document describes the components of VAX OPS5. It is a reference tool for writing VAX OPS5 programs.

If you want to use a VAXstation to create, run, and revise VAX OPS5 programs, you should also read the *VAX OPS5 Development Environment User's Guide*.

Revision/Update Information:	This document supersedes the *VAX OPS5 Reference Manual*, AA–EZ19B–TE.
Operating System and Version:	VMS Version 5.1 or higher and DECwindows for the VAX OPS5 Development Environment.
	VMS Version 5.0 or higher for the VAX OPS5 compiler and run-time system.
Software and Version:	VAX OPS5 Version 3.0

digital equipment corporation
maynard, massachusetts

First Printing, September 1985
Revised, February 1988
Revised, May 1989

ML–S1161

63451

Contents

Preface

Manual Objectives

This document provides VAX OPS5 programmers with a complete description of VAX OPS5 components.

If you want to use a VAXstation to create, run, and revise VAX OPS5 programs, you should also read the *VAX OPS5 Development Environment User's Guide*.

Intended Audience

The document is for readers who have a basic understanding of VAX OPS5. Some familiarity with the VMS operating system is helpful. For information concerning this system, refer to the section Associated Documents in this preface.

Structure of This Document

This document is divided into three parts and one appendix. Part I consists of four chapters that describe the VAX OPS5 components. Part II consists of two chapters that explain how to write programs in VAX OPS5. Part III consists of seven chapters that describe the VAX OPS5 operators, declarations, statements, actions, functions, commands, and support routines. The appendix tells you how to use the %INCLUDE compiler directive.

Associated Documents

- *VAX OPS5 User's Guide*
- *VAX OPS5 Version 3.0 Release Notes* (on-line)
- *The Artificial Intelligence Education Series: VAX OPS5* (self-paced instruction)
- *VAX OPS5 Development Environment User's Guide*
- *VMS DECwindows User's Guide*
- *VAX Architecture Handbook*
- *Introduction to VMS*
- *VMS DCL Dictionary*
- *Guide to Using VMS Command Procedures*
- *VMS Record Management Services Manual*

For a complete list of VMS software documents, see the *VMS Master Index.*

NOTE

In addition to the aforementioned VAX documents, the following text is also recommended: *Rule-based Programming with OPS5* by Thomas Cooper and Nancy Wogrin.

To order this text, write to:

Morgan Kaufmann Publishers, Inc.
P.O. Box 50490
Palo Alto, CA 94303–9953
Attn: Michael McClatchey

Or, you can phone the publisher at (415) 965–4081.

Document Conventions

The following conventions are used in this document:

Convention	Meaning
[ ]	Square brackets enclose items that are optional. For example: `[file-id]`
...	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: `action...` You can specify many of the constructs described in Part III with more than one argument. If you specify more than one argument with an operator, declaration, statement, action, function, or command, separate the argument values with any combination of spaces, tabs, and carriage returns. If you specify more than one argument with a support routine, separate the values with a comma and a space.
{ }	In format specifications, braces enclosing a horizontal list of items indicate items that are considered one unit of code. For example: `{scalar-attribute value}` In these cases, do not include the braces in your code. In examples of VAX OPS5 code, however, braces enclose conjunctions and specify element variables. In these cases, the braces must be included in the syntax.
{ }...	Braces followed by a horizontal ellipsis mean that you can repeat the enclosed unit of code one or more times. For example: `{attribute-name = field}...`
$\{\ \}$	Braces enclosing a vertical list of items indicate that you must choose one of the items. For example: $\left\{\begin{array}{l}\text{action}\\ \text{command}\end{array}\right\}$

Convention	Meaning
. . .	A vertical ellipsis in a figure or example indicates that not all the information the system displays is shown or that not all the information you should enter is shown.
UPPERCASE characters	DCL commands and qualifiers and the names of VAX OPS5 declarations, statements, actions, functions, commands, and support routines are printed in uppercase characters. However, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase.
lowercase characters	The arguments you must specify with DCL commands and VAX OPS5 operators, declarations, statements, actions, functions, commands, and support routines are printed in lowercase characters. However, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters.

blue-green ink	In examples, user input is printed in blue-green ink. For example:
	`OPS5> EXIT`
	`$`
	A carriage return is the implied terminator for user input at the end of command lines. If a control character or other type of terminator is required, it will be explicitly stated in the text.

decimal notation	All numeric values are represented in decimal notation.

Part I
VAX OPS5 Components

Part I of this manual provides information about the VAX OPS5 components.

Chapter 1 introduces the structure of VAX OPS5. Chapters 2, 3, and 4 describe working memory, productions, and the recognize-act cycle.

Chapter 1

Introduction to VAX OPS5

VAX OPS5 is used in the field of artificial intelligence for developing applications for expert systems and cognitive psychology. It is characterized by:

- A global data base
- Condition-action (or IF-THEN) rules programmed in the form of productions, which operate on the global data base
- Productions that are executed in an unspecified order
- Computation with symbolic expressions and numbers
- Simple syntax
- Knowledge representation

VAX OPS5 is an extended implementation of standard OPS5. VAX OPS5 programs are compiled using the VAX OPS5 compiler and executed using the VAX OPS5 run-time system. This manual provides information about VAX OPS5. For tutorial information, see the self-paced instruction (SPI) course *The Artificial Intelligence Education Series: VAX OPS5*. For information about how to use VAX OPS5 with the VMS operating system, see the *VAX OPS5 User's Guide*.

NOTE

In addition to the aforementioned VAX documents, the following text is also recommended: *Rule-based Programming with OPS5* by Thomas Cooper and Nancy Wogrin.

To order this text, write to:

Morgan Kaufmann Publishers, Inc.
P.O. Box 50490
Palo Alto, CA 94303–9953
Attn: Michael McClatchey

Or, you can phone the publisher at (415) 965–4081.

This chapter provides an overview of the structure of VAX OPS5 and describes VAX OPS5 atoms and program elements.

1.1 VAX OPS5 Structure

The VAX OPS5 system has two key components: a data base called working memory and productions that manipulate the data base. The run-time system uses a recognize-act cycle to process the contents of working memory and the productions.

Working memory, productions, and the recognize-act cycle are described in Chapters 2, 3, and 4, respectively.

1.1.1 Working Memory

Working memory is a global data base that stores elements that describe a problem. Each element can have a class name and a list of associated attributes and their values. The class name classifies the element according to the type of information the element contains. The attributes and their values describe the element's characteristics.

Suppose a VAX OPS5 program requires a data base that contains the following information about checks drawn on a bank account:

Checks

Number	**Amount**	**Date**
102	10.06	2 Nov 1988
103	22.45	14 Nov 1988
104	56.00	14 Nov 1988
108	13.10	25 Nov 1988

You can represent this information in working memory with four elements. To classify the elements, you can assign the class name CHECK to each element. You can use the symbols NUMBER, AMOUNT, and DATE to name the attributes that describe the elements' characteristics. For example, an element might look like:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^DATE 2 NOV 1988)
```

1.1.2 Productions

Productions operate on working memory. Each production has a name and consists of a condition part, called the left-hand side, and an action part, called the right-hand side. The left-hand side is a list of patterns called condition elements, with which working-memory elements are compared. The right-hand side is a list of actions that are executed when working-memory elements match the condition elements on the production's left-hand side. An action consists of an action name and its arguments, and usually manipulates the contents of working memory.

An example of a production follows:

```
(P OVERDRAWN-ACCOUNT
   (ACCOUNT ^NUMBER <ID> ^BALANCE <BALANCE>)
   (CHECK ^ACCOUNT-NUMBER <ID> ^AMOUNT > <BALANCE>)
  -->
   (WRITE (CRLF) (CRLF) |Your account is overdrawn.|)
   (HALT))
```

The name of this production is OVERDRAWN–ACCOUNT. The left-hand side consists of two condition elements:

```
(ACCOUNT ^NUMBER <ID> ^BALANCE <BALANCE>)

(CHECK ^ACCOUNT-NUMBER <ID> ^AMOUNT > <BALANCE>)
```

The right-hand side contains two actions—a WRITE action that displays a message and a HALT action that stops program execution.

1.1.3 Recognize-Act Cycle

The recognize-act cycle consists of four steps:

1. Recognize matches
2. Select a match
3. Act (execute the selected production)
4. Go to step 1

During the match step, the system compares working-memory elements with the condition elements on the left-hand side of each production. When working-memory elements match the condition elements, the production is ready for execution. The system selects one of the ready productions, executes the actions on its right-hand side, and begins the cycle again.

1.2 Atoms

The unit of data in a VAX OPS5 program is called an atom. There are three types of atoms: symbolic, integer, and floating point.

1.2.1 Symbolic Atoms

A symbolic atom is one that does not have a numeric value. For example:

```
c

CHECK

?-c

10-14
```

1.2.2 Integer Atoms

Integer atoms consist of the following:

- An optional plus or minus sign
- One or more decimal digits
- An optional decimal point

The following are examples of integer atoms:

```
2
20.
-20
-2.
```

The valid range for integer atoms is –2**29 to 2**29–1.

1.2.3 Floating-Point Atoms

A floating-point atom is composed of the following:

- An optional plus or minus sign
- Zero or more decimal digits
- A decimal point
- One or more decimal digits after the decimal point
- An optional exponent

An exponent consists of the letter e followed by a signed or unsigned integer and represents a power of 10 by which a preceding number is to be multiplied. For example, e–8 represents the value 10 raised to the power –8.

NOTE

A floating-point atom must include a decimal point followed by a digit, an exponent, or both.

The following are examples of floating-point atoms:

```
 0.0
  .25
10.05e-14
-5.e10
```

Floating-point atoms are implemented as VAX F_floating data. The valid range for floating-point atoms is .29e–38 to 1.7e38. The precision is approximately seven decimal digits. For information about the VAX floating data types, see the *VAX Architecture Handbook.*

1.2.4 Quoted Atoms

The following characters have a specific meaning in the VAX OPS5 syntax:

- Escape character
- Control characters
- Space
- Tab
- Parentheses (( ))

- Braces ({ })
- Circumflex (^)
- Semicolon (;)

To include these characters in an atom, "quote" the atom by enclosing it in vertical bars (| |). The text between the two vertical bars is considered one symbolic atom. For example, the compiler and run-time system recognize THIS IS AN ATOM to be four atoms separated by spaces. However, in the following example, they recognize THIS IS AN ATOM to be one atom:

```
|THIS IS AN ATOM|
```

Quoted atoms are symbolic atoms; therefore, | 1.2 | is a symbol, not a floating-point number, and arithmetic operations cannot be performed on it.

The opening and closing quotes must appear on the same line in the code. They can enclose as many characters as are allowed in an atom's print name. (The maximum number of characters an atom's print name can consist of is 256.)

The circumflex (^) is the VAX OPS5 attribute operator, but in the following example vertical bars enclose the atom, so it is treated as any ordinary character.

```
|^NUMBER|
```

If the atom you enclose in vertical bars includes a vertical bar, double the vertical bar. For example:

```
|This is a vertical bar -- ||.|
```

When the atom is displayed or printed, it includes only one vertical bar:

```
This is a vertical bar -- |.
```

The VAX OPS5 compiler and run-time system do not distinguish between uppercase and lowercase characters unless the characters appear inside vertical bars. For example, the word CHECK has the same meaning in any of the following forms:

```
CHECK
ChecK
check
```

However, the following examples represent different values:

```
|CHECK|
|Check|
```

1.3 Program Elements

VAX OPS5 programs consist of declarations, executable statements, and optional comments.

1.3.1 Declarations

Declarations are units of code, each enclosed in parentheses, that define the attributes and external routines used in a program. Attribute declarations are described in Section 2.3. External routine declarations are described in Chapter 8. Declarations must precede any other kind of statement.

1.3.2 Executable Statements

Executable statements are units of code, each enclosed in parentheses, that perform some type of operation. An executable statement can be a startup statement, catcher, or production. Startup statements and catchers are described in Sections 5.1 and 5.9, respectively. Productions are described in Chapter 3.

To improve the readability of a program, you can format units of code by using spaces, tabs, and new-line characters.

1.3.3 Comments

A VAX OPS5 program can contain comments. A comment starts with a semicolon and finishes at the end of the same line. For example:

```
;This is a comment.
```

If you want comment text to extend over more than one line, you must start each line with a semicolon.

The compiler ignores the text of comments. Therefore, comments can be used at the start of a program, before the attribute declarations; they can contain any ASCII character and can appear anywhere a space character is valid.

If you use a semicolon as part of an atom, you must enclose the atom in vertical bars to ensure that the semicolon is not treated as the start of a comment.

Chapter 2
Working Memory

Working memory is a global data base of information representing the problem a VAX OPS5 program is to solve. The information is stored in elements, which are grouped into classes. Elements storing similar information can be grouped in the same class.

Working memory can contain several classes of elements, and each class can have more than one element. An example of a class that contains only one element is a class representing the opening balance of a checking account. An example of a class that contains more than one element is a class representing the records written in a checkbook register.

Working memory is dynamic. As a VAX OPS5 program is executed, elements are added, deleted, and modified continually.

2.1 Working-Memory Elements

A working-memory element is a sequence of atoms that represents an object or concept. Each atom is stored in a field that you can label with an attribute name. You can specify a working-memory element, using a combination of the following:

- A class name
- A list of scalar attributes and their values
- A vector attribute and its value

The class name identifies the element's class, and the attributes and their values describe the element's characteristics. The value of each scalar attribute is an atom. The value of a vector attribute can be one or more atoms. The format for specifying a working-memory element is as follows:

[class-name] [{scalar-attribute value}...] [vector-attribute value]

Consider the following element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED YES
       ^DATE 2 NOV 1988)
```

This statement represents a working-memory element of the class CHECK. The attributes ^NUMBER, ^AMOUNT, ^COUNTED, and ^DATE represent four characteristics of the element. The values of the scalar attributes ^NUMBER, ^AMOUNT, and ^COUNTED are the atoms 102, 10.06, and YES, respectively. The value of the vector attribute ^DATE is the list of atoms 2 NOV 1988.

2.1.1 Class Name

A class name is a symbol that identifies a group of similar elements. Elements that have the same class name have the same attributes, but the values of the attributes are different. For example, the following elements have the class name CHECK:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
       ^DATE 2 NOV 1988)

(CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO
       ^DATE 14 NOV 1988)
```

2.1.2 Attributes

An attribute consists of the attribute operator (^), followed by an attribute name, which describes the element's characteristics.

Suppose you want to specify a working-memory element that has the class name CHECK, a number, an amount, and a count status. You can specify these characteristics with the attribute names NUMBER, AMOUNT, and COUNTED as follows:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
```

You can use the same attribute name in more than one element even if the elements have different class names. For example, the attribute ^NUMBER is used in both of the following elements:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
(TRANSACTION ^NUMBER 2560 ^TYPE DEPOSIT)
```

An attribute name can also be an integer. The integer indicates the field of the working-memory element in which the attribute's value is stored or is to be stored (see Section 2.2.2). (Using integers as attribute names is not recommended, because it can cause confusion.)

You do not have to specify all the attributes associated with a class name. If the class name CHECK is associated with the attributes ^NUMBER, ^AMOUNT, and ^COUNTED, you can specify:

```
(CHECK ^AMOUNT 10.06)
```

This element has the class name CHECK and the attribute ^AMOUNT, whose value is the atom 10.06.

The next two subsections explain the difference between the two types of attributes.

2.1.2.1 Scalar Attributes

The value of a scalar attribute is an atom (see Section 1.2). For example:

```
^NUMBER 102
```

The following example shows a working-memory element whose class name is CHECK and whose scalar attributes are ^NUMBER, ^AMOUNT, and ^COUNTED:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
```

The values of the attributes are the atoms 102, 10.06, and NO, respectively.

The maximum number of scalar attributes you can specify for an element is 255.

2.1.2.2 Vector Attributes

The value of a vector attribute is a list of one or more atoms. For example:

```
^DATE 2 NOV 1988
```

Consider the following working-memory element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
```

Suppose you add the vector attribute ^DATE with the list of atoms 2 NOV 1988:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
       ^DATE 2 NOV 1988)
```

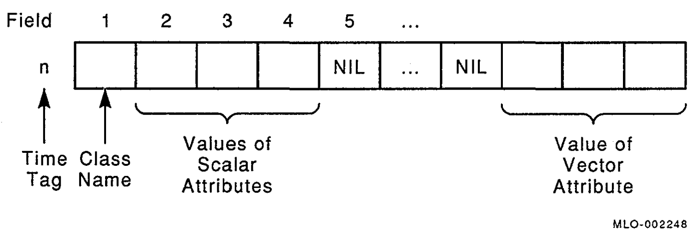
This working-memory element has the class name CHECK; the scalar attributes ^NUMBER, ^AMOUNT, and ^COUNTED; and the vector attribute ^DATE.

The number of atoms that you specify for a vector attribute can vary during a program's execution. The value of a vector attribute can consist of a maximum of 127 atoms.

2.2 Internal Representation of Working-Memory Elements

The internal representation of a working-memory element includes a time tag and one or more atoms, which represent the element's class and attribute values. Figure 2–1 illustrates how the atoms are stored in fields.

Figure 2–1: Internal Representation of a Working-Memory Element



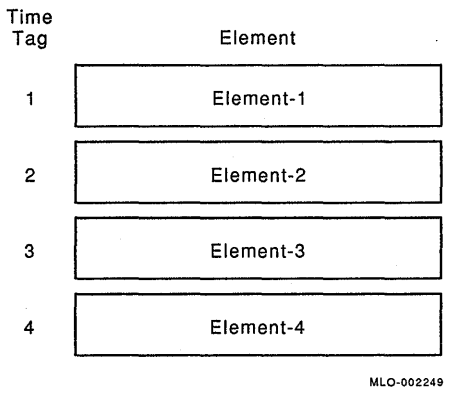
NOTE

NIL is stored in the fields that are not assigned atoms.

2.2.1 Time Tags

Time tags are integers that the run-time system uses to determine recency during conflict resolution (see Section 4.2.1.2). The run-time system assigns a unique time tag to each element in working memory. The element with the largest time tag is the most recent. Figure 2–2 illustrates how the run-time system assigns time tags.

Figure 2–2: Time Tag Representation



Element–4 has the largest time tag and therefore is the most recent.

In addition to the run-time system using time tags to determine recency, you can use time tags as arguments for some VAX OPS5 commands (see Chapter 12 and the *VAX OPS5 User's Guide*).

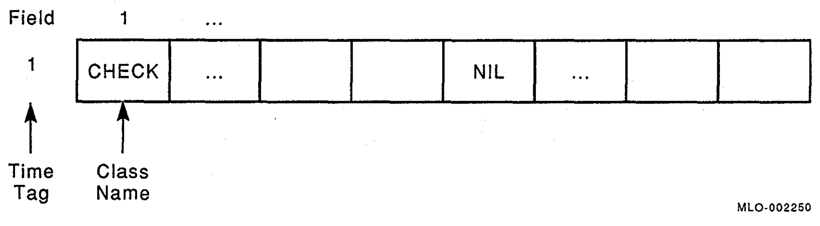
When you delete an element from working memory, you also delete the element's time tag, which is not used again during the program's execution. Likewise, if you modify an element, it is assigned a new time tag. (When you modify an element, the run-time system deletes that element from working memory and adds a revised element.)

2.2.2 Storing the Class Name and Attribute Values

The first field of the structure that stores a working-memory element is reserved for the element's class name. If an element does not have a class name, the run-time system places NIL in the first field. Figure 2–3 shows how the run-time system stores the class name for the following element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
       ^DATE 2 NOV 1988)
```

Figure 2–3: Storing the Class Name

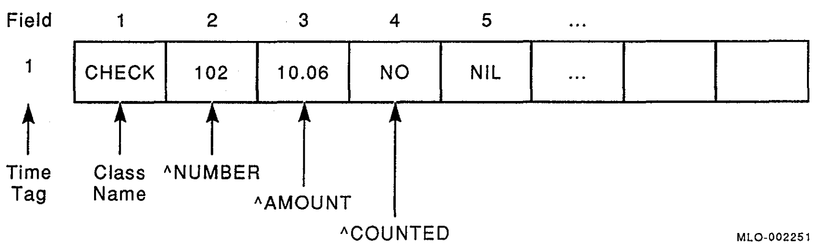


The compiler assigns fields to the names of the scalar attributes when the attribute names are declared (see Section 2.3). An attribute's field stores that attribute's value. Consider the following working-memory element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
```

Figure 2–4 illustrates the internal representation when the compiler assigns field 2 to NUMBER, field 3 to AMOUNT, and field 4 to COUNTED.

Figure 2–4: Storing the Values of Scalar Attributes

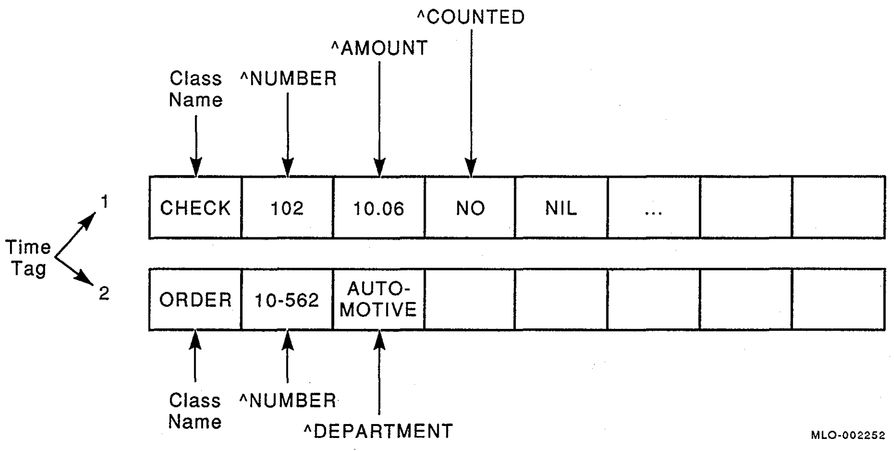


The field assigned to each attribute name is global, that is, the attribute name refers to the same field for each element class in which the attribute name appears. Consider the following elements:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
(ORDER ^DEPARTMENT AUTOMOTIVE ^NUMBER 10-562)
```

These elements have different class names but share the scalar attribute ^NUMBER. Suppose field 2 in the first element corresponds to the attribute name NUMBER. Since the attribute name refers to the same field for both elements, the value of the attribute ^NUMBER is placed in field 2 for both elements, although the value in each element can be different. Figure 2–5 illustrates this.

Figure 2–5: Shared Attributes



You can refer to a field in an element's structure directly by specifying the number of the field with the attribute operator. For example, to refer to the atom stored in the second field, you can specify:

```
^2
```

However, using the attribute operator with an integer is not recommended because you might refer to the wrong atom. For example, the compiler might assign the fields 2, 3, and 4 to the attribute names NUMBER, AMOUNT, and COUNTED, respectively. Suppose you want to refer to the attribute ^NUMBER. If you specify the attribute operator with 3, you refer to the value of the attribute ^AMOUNT, not the value of ^NUMBER. However, if you use the attribute name, you will always refer to the correct atom. Using attribute names also makes debugging and maintaining programs easier.

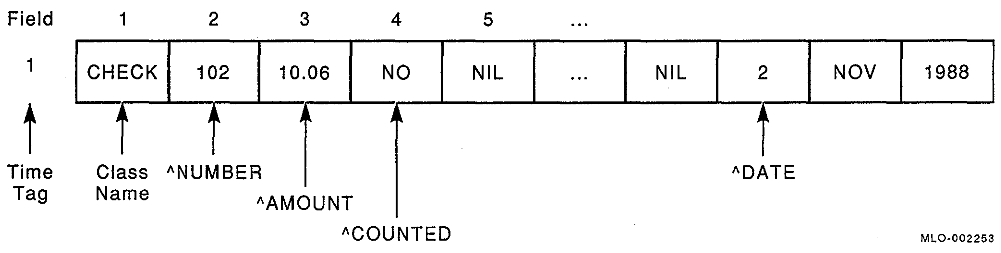
The compiler reserves field 256 of each working-memory element for the start of the vector attribute, if there is one (see Section 2.3.3). Assigning the value of vector attributes to this location eliminates the chance of the run-time system writing over scalar-attribute fields. The predefined field stores the first atom of the vector attribute's value. The rest of the atoms are stored sequentially in the remaining fields.

Consider the following working-memory element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
       ^DATE 2 NOV 1988)
```

Figure 2–6 shows this element's internal representation.

Figure 2–6: Storing the Value of a Vector Attribute



2.3 Declarations

Attribute names and external routines must be declared using the LITERALIZE, LITERAL, VECTOR–ATTRIBUTE, or EXTERNAL declaration, which must appear before any other kind of statement. The following sections explain how to use these declarations. For detailed descriptions of the declarations, see Chapter 8.

2.3.1 LITERALIZE Declarations

Use the LITERALIZE declaration to do the following:

- Associate a class with a list of attribute names
- Tell the compiler to assign unique fields to the specified attribute names

An example of a LITERALIZE declaration follows:

```
(LITERALIZE CHECK
            NUMBER
            AMOUNT
            COUNTED)
```

This declaration associates the class name CHECK with the attribute names NUMBER, AMOUNT, and COUNTED. The declaration lets the program refer to working-memory elements that have the class name CHECK and any combination of the attributes whose names are listed.

This declaration also tells the compiler to assign fields to the specified attribute names. The compiler does not necessarily assign fields in the order in which you write the attribute names. If you use the same attribute name in more than one LITERALIZE declaration, the same field position will be assigned to each.

You can find out which field the compiler has assigned to an attribute name by calling the LITVAL function in a right-hand-side action (see Section 5.6.2). Consider the preceding LITERALIZE declaration, and suppose the compiler assigns field 2 to NUMBER. The following call to the LITVAL function returns 2:

```
(LITVAL NUMBER)
```

2.3.2 LITERAL Declarations

You can explicitly assign fields to attribute names, using the LITERAL declaration. For example:

```
(LITERAL NUMBER = 2
         AMOUNT = 4
         COUNTED = 7)
```

This LITERAL declaration assigns field 2 to NUMBER, field 4 to AMOUNT, and field 7 to COUNTED.

If a program contains both LITERAL and LITERALIZE declarations, the compiler processes the LITERAL declarations first, regardless of the order in which you specify the declarations. The compiler uses the field assignments for the attribute names specified in both LITERAL and LITERALIZE declarations. If the compiler cannot use the assignments specified by a LITERALIZE declaration, it displays the following error message:

```
%OPSCOMP-W-LITCLASH, Literal value clash involving AAAAAA in
literalize declaration -- old value kept
```

You should use LITERALIZE declarations rather than LITERAL declarations for two reasons. First, when you use a LITERAL declaration, you might inadvertently assign the same field to two different attribute names. For example, suppose you specified the following LITERAL declarations:

```
(LITERAL NUMBER = 2
         AMOUNT = 4
         COUNTED = 5
         DATE = 6)

(LITERAL NAME = 3
         COUNT = 4)
```

If you used the attributes ^COUNT and ^AMOUNT in the same working-memory element, the program would overwrite the values of these attributes unless you changed one of the declarations, since field 4 would be assigned to the names of both attributes.

Second, LITERAL declarations do not associate a class with attribute names. Therefore, when you use LITERAL declarations, the run-time system displays working-memory elements in lists rather than in an attribute-value-pair format, making the debugging process more difficult.

2.3.3 VECTOR–ATTRIBUTE Declarations

Use the VECTOR–ATTRIBUTE declaration to assign field 256 to the name of a vector attribute. The run-time system stores the atoms of the attribute's value, starting in that field. The following example declares the vector attribute named DATE:

```
(VECTOR-ATTRIBUTE DATE)
```

You can declare all the vector attributes of a program in one VECTOR–ATTRIBUTE declaration or declare each vector attribute separately.

After you have declared a vector attribute, you can specify the name of that attribute in a LITERALIZE declaration. For example:

```
(LITERALIZE CHECK
            NUMBER
            AMOUNT
            COUNTED
            DATE)
```

This declaration associates the class name CHECK with the attribute names NUMBER, AMOUNT, COUNTED, and DATE. Only one vector attribute can be declared for a class.

2.3.4 EXTERNAL Declarations

If your VAX OPS5 program contains calls to an external routine, which is a routine written in a language other than VAX OPS5, you must declare it using an EXTERNAL declaration, which identifies it to the VAX OPS5 compiler.

The EXTERNAL declaration is described in Chapter 8.

Chapter 3
Productions

Productions are the condition-action statements of a VAX OPS5 program. If data in working memory matches the conditions in a production, the run-time system can execute the production's actions. Each production consists of:

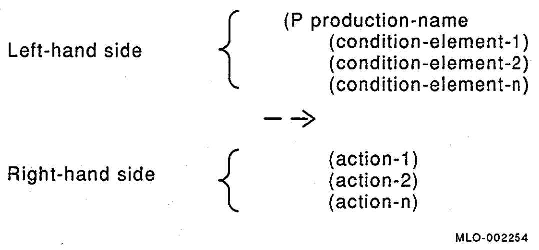
- A production name
- A left-hand side (LHS)
- An arrow (– –>)
- A right-hand side (RHS)

The left-hand side of a production consists of one or more condition elements, which contain the patterns that working-memory elements must match. Each condition element must be enclosed in parentheses.

The right-hand side consists of one or more actions. Actions instruct the run-time system to perform operations, such as add, remove, or modify working-memory elements. Each action must be enclosed in parentheses.

Figure 3–1 illustrates the structure of a production.

Figure 3–1: Production Format



```
(P production-name
    (condition-element-1)
    (condition-element-2)
    (condition-element-n)
-->
    (action-1)
    (action-2)
    (action-n)
```

Productions must be enclosed in parentheses. The P, which follows the opening parenthesis, signifies that the following code is a production. The production name distinguishes the production from other productions in a program.

The left-hand side of the production is separated from the right-hand side by an arrow, created by typing two dashes and a greater-than sign (– –>).

An example of a production follows:

```
(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are| <VALUE> |checks dated|
                          <DAY> <MONTH> <YEAR> (CRLF)))
```

The name of this production is COUNTED–CHECKS. The left-hand side consists of three condition elements:

```
(REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>)

-(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)

(COUNT ^VALUE <VALUE>)
```

The braces ({ }) enclosing the first and third condition elements define element variables <REPLY> and <COUNTER>. Section 3.2.2 explains how to specify element variables.

The minus sign preceding the second condition element indicates that it is negative. Section 3.2 explains positive and negative condition elements.

The right-hand side contains four actions: two REMOVE actions, a MAKE action, and a WRITE action.

You can list productions in any order without affecting the way the program is executed. However, if you group productions that work toward the same goal, the program is easier to read and maintain.

3.1 Production Name

A production name is a symbol that identifies a production. The name of each production in a program must be unique. You can name a production with any symbol except NIL. (The run-time system uses NIL to identify working-memory elements created by a VAX OPS5 command.)

3.2 Left-Hand Side—Condition Elements

The left-hand side of a production contains one or more condition elements. The run-time system compares the atoms in working-memory elements with corresponding patterns in condition elements.

Specifying a condition element is similar to specifying a working-memory element. Use any combination of the following:

- A class name
- A list of scalar attributes and their values
- A vector attribute and its value

For descriptions of these components, see Sections 2.1.1 and 2.1.2.

Condition elements can be positive or negative. The first condition element must be positive. A left-hand side can have a maximum of 32 positive condition elements and unlimited negative condition elements.

A production is ready for execution when working-memory elements match all the production's positive condition elements, but none matches its negative condition elements. (See Section 4.1 for a description of the match phase of the recognize-act cycle.) For example, suppose the left-hand side of a production contains the following condition elements:

```
(REPLY ^DATE STOP)
-(CHECK ^DATE 14 NOV 1988)
(COUNT ^VALUE 10)
```

The production is ready for execution when working-memory elements match the first and third condition elements, but none matches the second condition element.

The sections that follow explain how you can specify components in a condition element and how to use element variables.

3.2.1 Specifying Condition-Element Components

The values you specify in the components of a condition element can be constant atoms, variables, or function calls. They can be preceded by a predicate, indicating the comparison operation to be performed (equal to, greater than, and so on). Combinations of comparison operations for an attribute can be built using conjunctions and disjunctions, which are similar to AND and OR operators.

If a symbol in a working-memory element and a symbol in a condition element are composed of the same sequence of characters, the symbols match. A number in a working-memory element and a number in a condition element match if the difference between the two numbers is zero and both numbers are the same type (integer or floating point).

The run-time system compares each atom in a working-memory element using the comparison operation specified in the corresponding component. The system uses the following rules for each comparison:

- If the component specifies a class name, the system compares the first atom in the working-memory element with that name.
- If the component specifies a scalar attribute and value, the system compares that attribute's value with the value specified in the component.
- If the component is a vector attribute and a value, the system compares the first atom of that attribute's value with the first atom of the value specified in the component. The system continues comparing atoms until the value is exhausted.

  The match phase of the recognize-act cycle is described in Section 4.1.

Suppose working memory contains an element in which fields 2, 3, and 4 are assigned to the attribute names NUMBER, AMOUNT, and COUNTED, respectively:

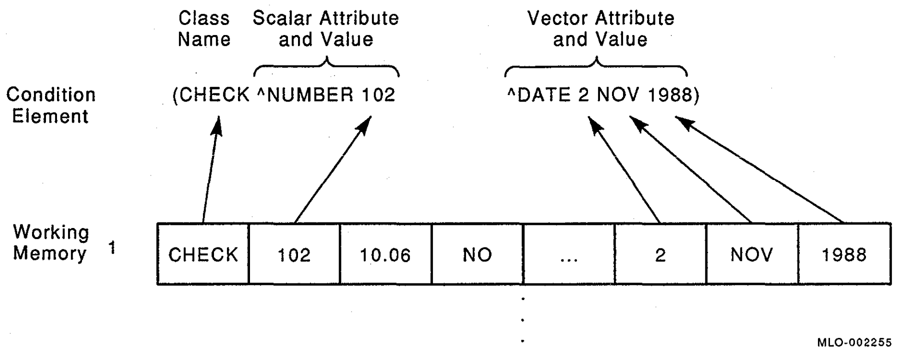
```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
       ^DATE 2 NOV 1988)
```

Now consider the following condition element:

```
(CHECK ^NUMBER 102 ^DATE 2 NOV 1988)
```

Figure 3–2 illustrates how the run-time system compares the atoms in the preceding working-memory element with the values of this condition element's components.

Figure 3–2: Matching Atoms in a Working-Memory Element with Component Values



The remaining atoms in the working-memory element are ignored and do not influence the match process.

The following subsections explain how to specify component values, using constants, variables, predicates, conjunctions, disjunctions, function calls, and the quote operator.

3.2.1.1 Constants

Constants can be symbols, integers, or floating-point numbers.

3.2.1.2 Variables

A variable is a symbol enclosed in angle brackets (< >). An example is <NUMBER>.

NOTE

> In VAX OPS5, the symbol <=> represents the same-type operator. You cannot use it as a variable.

Variables refer to unknown atoms in a working-memory element. The first time a variable is used in a production, the variable is bound to the atom in the working-memory element matching the condition element. All occurrences of that variable in that production represent the same atom. For example, suppose the left-hand side of a production contains the following condition elements:

```
(REPLY ^DATE <DAY> <MONTH> <YEAR>)

-(CHECK ^DATE <DAY> <MONTH> <YEAR>)
```

If the run-time system finds a match for the first condition element, the system binds the variable <DAY> to the first atom of the vector attribute ^DATE, the variable <MONTH> to the second atom, and the variable <YEAR> to the third atom. Suppose the variable <DAY> is bound to atom 2. Then the variable <DAY> in the second condition element must represent atom 2.

3.2.1.3 Predicates

Predicates are operators that can precede values (constants and variables) in condition-element components. Predicates test the atoms in working-memory elements and produce a match if the atoms meet specific conditions. The predicates are:

=	Same type as and equal to
<>	Not same type as or not equal to
<=>	Same type as
<	Same type as and less than
<=	Same type as and less than or equal to
>	Same type as and greater than
>=	Same type as and greater than or equal to

Every value specified in a condition element is preceded by a predicate, either implicitly or explicitly. Values that you specify without a predicate are implicitly preceded by the equal operator. For example, the following components are equivalent:

```
^NUMBER 102
```

```
^NUMBER = 102
```

The equal operator is the only predicate that can precede the first occurrence of a variable, because the first time a variable occurs, it is bound to an atom. The rest of the predicate operators must be specified with either an atom or a variable bound to an atom. Suppose the left-hand side of a production consists of the following condition elements:

```
(REPLY ^DATE <DAY> <MONTH> <YEAR>)
```

```
(CHECK ^DATE > <DAY>)
```

If the variable <DAY> in the first condition element is bound to atom 10, the run-time system uses 10 to find working-memory elements that match the second condition element. A working-memory element that has the class name CHECK and a vector attribute ^DATE whose first atom is an integer greater than 10 produces a match.

You can use the equal (=), not-equal (<>), and same-type (<=>) operators with any type of atom or a variable bound to any type of atom. The rest of the operators must be specified with integers, floating-point numbers, or variables bound to integers or floating-point numbers, or with calls to functions that return integers or floating-point numbers.

Comparisons of two atoms fail if they are not the same type. For example, a match on the component ^NUMBER < <NUMBER> will fail unless the atom in a working-memory element is an integer and the variable <NUMBER> is bound to an integer, or the element is a floating-point number and the variable <NUMBER> is bound to a floating-point number. (The number in the working-memory element must also be less than the number bound to the variable for the match to succeed.)

See Chapter 7 for descriptions of the predicate operators.

3.2.1.4 Conjunctions

A conjunction is a pattern of conditional tests all of which must be true of an atom in a working-memory element. A conjunction is similar to a logical AND.

You specify a conjunction by enclosing the list of conditional tests in braces ({ }).

A conjunction is useful for binding a variable to an atom that satisfies one or more conditional tests. For example, if you want an integer between 102 and 105, you can specify the following condition element, which contains the appropriate conjunction:

```
(CHECK ^NUMBER { > 102 < 105 })
```

If you also want to bind the value of the attribute ^NUMBER to a variable, you can use the following conjunction:

```
(CHECK ^NUMBER {<NUMBER> > 102 < 105 })
```

If working memory contains an element that has the class name CHECK and an attribute ^NUMBER whose value is 103 or 104, a match results. The run-time system then binds the variable <NUMBER> to that value.

You can also use a conjunction as a placeholder by specifying the braces without conditional tests. A placeholder can be used to skip over atoms in a vector attribute's value. For example:

```
(CHECK ^DATE { } <MONTH>)
```

A working-memory element that has the class name CHECK and a vector attribute whose second atom is bound to <MONTH> matches this condition element. The values of the other attributes do not affect the match.

You can also use a placeholder when you specify a condition element without using attribute names. For example, if you had used a LITERAL declaration to assign fields 2, 3, and 4 to the attributes ^NUMBER, ^AMOUNT, and ^COUNTED you might specify:

```
(CHECK { } { } YES)
```

A working-memory element whose fourth field contains the atom YES matches this condition element.

3.2.1.5 Disjunctions

A disjunction is a pattern containing a list of constant atoms. Only the atoms specified in the list can match the pattern. A disjunction is similar to a logical inclusive OR.

Specify a disjunction by enclosing the list of constant atoms between double angle brackets (<< >>). The following condition element contains a disjunction:

```
(CHECK ^NUMBER << 103 105 108 >>)
```

If working memory contains an element whose class name is CHECK and whose ^NUMBER attribute has the value 103, 105, or 108, a match results.

The atoms you specify in a disjunction are not evaluated. Therefore, operators and variables are recognized as symbols. Consider the following disjunction:

```
<< ^NUMBER <NUMBER> >>
```

The run-time system recognizes the circumflex (^), NUMBER, and <NUMBER> to be symbols, not an attribute operator, attribute name, and variable.

3.2.1.6 Function Calls

You can represent a component value with a call to the COMPUTE function or an external function. Function calls perform an operation and return one or more atoms when the operation is complete. A function call is a function name and its arguments enclosed in parentheses. The format for specifying a function call is:

(function-name argument–1 argument–2...)

For example:

```
(COMPUTE 1 + <VALUE>)
```

COMPUTE is the name of the function, and the rest of the values are the function's argument values.

NOTE

> If a function requires no arguments, you must still enclose the function name in parentheses.

Function calls can include variables, but the variables must be bound to atoms, that is, the variables must have been used in a previous condition element or previously in the same condition element.

The COMPUTE function evaluates an arithmetic expression and returns the result. For example:

```
(P UPDATE-COUNTER
     (CURRENT-TASK ^NAME COUNTING ^COUNTED <C>)
     (COUNTER ^COUNT (COMPUTE <C> - 1))
   -->
                    .
                    .
                    .
```

The first condition element binds the variable <C> to a number in a working-memory element. A working-memory element that has the class name COUNTER, and an attribute ^COUNT whose value is a number one less than the number to which the variable <C> was originally bound, will match the second condition element. (For more information about using the COMPUTE function, see Section 5.7.)

You can also represent a component value with a call to an external function. An external function is a function written in a language other than VAX OPS5.

If a condition element contains a call to an external function, you must declare the function at the beginning of the program with the EXTERNAL declaration (see Chapter 8).

The following condition element contains the function call (SQUARE_ROOT <VARIANCE>):

```
(MEANSD ^MEAN <MEAN> ^STDDEVIATION (SQUARE_ROOT <VARIANCE>))
```

For more information about calling external functions, see the *VAX OPS5 User's Guide*.

3.2.1.7 Quote Operator

In condition elements, you can quote component values so that they are not evaluated. This allows you to use any symbol, operator, variable, or function call as a constant atom.

To quote a value, precede it with the quote operator (//). Using this operator is similar to enclosing an atom in vertical bars (see Section 1.2.4). For example:

```
(CHECK ^NUMBER // <NUMBER>)
```

The atom <NUMBER> in a working-memory element will match the symbol <NUMBER>. If you do not use the quote operator, an atom will match the atom bound to the variable <NUMBER>.

You should quote function calls when you create working-memory elements that are used by the BUILD action (see Section 5.11). An example of a quoted call to the SUBSTR function follows:

```
// (SUBSTR <BUILD-PRODUCTION> 2 INF)
```

The function call is treated like a list of atoms that can be placed in a working-memory element.

3.2.2 Specifying Element Variables

An element variable is a variable that is bound to a working-memory element. To specify an element variable, enclose the variable and a positive condition element in braces ({ }). For example:

```
{ <COUNTER> (COUNT ^VALUE <VALUE>) }
```

This can also be written with the variable after the condition element:

```
{ (COUNT ^VALUE <VALUE>) <COUNTER> }
```

When a working-memory element matches the condition element, the variable is bound to that working-memory element. In the preceding example, the variable <COUNTER> is bound to the working-memory element that matches the condition element:

```
(COUNT ^VALUE <VALUE>)
```

The CBIND action also binds an element variable to a working-memory element (see Section 5.6.3).

On a production's left-hand side, any element variable can be specified only once. However, a unique element variable can be specified for each positive condition element in the left-hand side of a production.

3.3 Right-Hand Side—Actions

The right-hand side of a production consists of one or more actions. Actions perform the following operations:

- Modify working memory
- Save and restore the state of working memory and the conflict set
- Stop program execution
- Bind variables
- Manipulate files

- Write output
- Control loops
- Add productions to executing programs
- Call external subroutines

An action includes an action name and its arguments enclosed in parentheses. The format for specifying an action is:

(action-name argument–1 argument–2...)

For example:

```
(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
            ^DATE 2 NOV 1988)
```

In this example, MAKE is the name of an action, and the rest of the values are the action's argument values.

NOTE

If an action does not require arguments, you must still enclose the action name in parentheses.

Most VAX OPS5 actions require at least one argument. An argument consists of an optional attribute (a "^" followed by the attribute name) and the attribute value.

You can represent argument values with atoms, variables bound to atoms, or function calls that evaluate to atoms. Sections 3.3.1 and 3.3.2 explain how to use variables and function calls.

For information about how to use actions in a VAX OPS5 program, see Chapter 5. For a detailed description of each action, see Chapter 10.

3.3.1 Variables

You can use a variable to represent an argument value in an action if the variable is bound to an atom. A variable can be bound to an atom in a condition element or in a BIND action. Section 3.2.1.2 explains how to bind variables in condition elements. Section 5.6 explains how to bind variables, using the BIND action.

The OPENFILE action opens a file for input or output and associates a file identification name with that file. The following OPENFILE action opens the file whose specification is bound to the variable <FILE–SPEC>:

```
(OPENFILE CHECKSI <FILE-SPEC> IN)
```

3.3.2 Function Calls

You can represent an argument value with a call to a VAX OPS5 function or an external function. The format for function calls is described in Section 3.2.1.6.

Consider the following MAKE command:

```
(MAKE REPLY ^DATE (ACCEPTLINE))
```

First, the run-time system evaluates the call to the ACCEPTLINE function. The MAKE command then uses the value returned by the ACCEPTLINE function to create a working-memory element.

You can also represent an argument value with a call to an external function. An external function is a function written in a language other than VAX OPS5.

If an action contains a call to an external function, you must declare the function at the beginning of the program with the EXTERNAL declaration (Chapter 8).

The following BIND action contains the function call (SQUARE_ROOT <VARIANCE>):

```
(BIND <STD_DEVIATION> (SQUARE_ROOT <VARIANCE>))
```

For more information about calling external functions, see the *VAX OPS5 User's Guide*.

3.3.3 Element Designators

An element designator is either an element variable that labels a condition element, or an integer that refers to the position of a condition element on the left-hand side of a production. The MODIFY and REMOVE actions and the SUBSTR function require an argument whose value is an element designator.

Element variables are bound to working-memory elements during the match phase of the recognize-act cycle or by CBIND actions (see Section 5.6.3). When you use an element variable in an action or function call, the variable refers to the working-memory element to which it is bound. Likewise, when an action or function uses an integer designator, the designator refers to the working-memory element that matches the condition element indicated by that integer.

The following production uses three element variables:

```
(P FIND-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
     { <CHECK>
       (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
              ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                          |for $| <AMOUNT>
                          |dated| (SUBSTR <REPLY> DATE INF))
     (MODIFY <CHECK> ^COUNTED YES)
     (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The element variable <CHECK> is bound to the working-memory element that matches the condition element:

```
(CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
       ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>)
```

The element variable <COUNTER> is bound to the working-memory element that matches the condition element:

```
(COUNT ^VALUE <VALUE>)
```

The MODIFY actions use the element variables to change the working-memory elements bound to those variables.

This production could also use integers as element designators:

```
(P FIND-CHECKS
     (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>)
     (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
            ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>)
     (COUNT ^VALUE <VALUE>)
   -->
     (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                          |for $| <AMOUNT>
                          |dated| (SUBSTR 1 DATE INF))
     (MODIFY 2 ^COUNTED YES)
     (MODIFY 3 ^VALUE (COMPUTE 1 + <VALUE>)))
```

In this version of the production, the MODIFY actions are specified with the element designators 2 and 3. When an element designator is integer n, the designator refers to the working-memory element that matches the nth positive condition element. Therefore, the designator 2 refers to the working-memory element that matches the condition element:

```
(CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
       ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>)
```

The designator 3 refers to the working-memory element that matches the condition element:

```
(COUNT ^VALUE <VALUE>)
```

NOTE

Use element variables for element designators whenever possible. When you use an integer element designator, it is more difficult to keep track of the working-memory element to which the designator refers.

Chapter 4

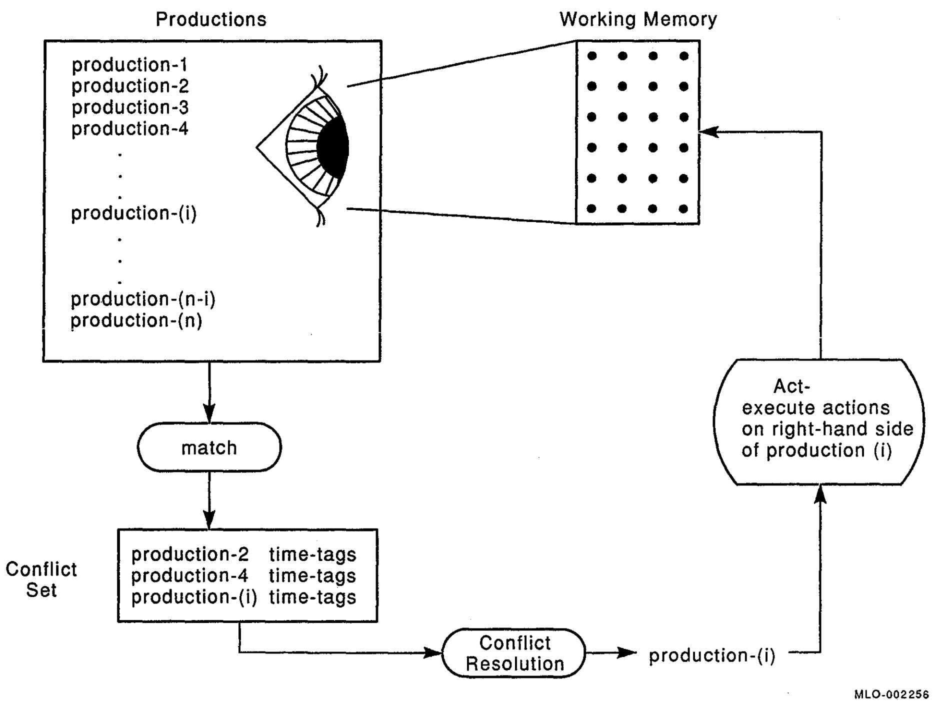
Recognize-Act Cycle

The VAX OPS5 run-time system uses a recognize-act cycle (Figure 4–1) to execute VAX OPS5 programs. The cycle consists of the following steps:

1. Match—Examines the current contents of working memory to locate all working memory elements that satisfy the condition elements in the left-hand sides of the program's productions. The productions whose left-hand sides are satisfied are placed in a list called the conflict set.
2. Conflict resolution—Selects one production from the conflict set. If the conflict set is empty (because no left-hand side has been satisfied) the program halts.
3. Act—Executes the actions on the right-hand side of the selected production. If the right-hand side contains the HALT action, the program halts.
4. Go to step 1.

This chapter describes the steps of the cycle.

Figure 4–1: Recognize-Act Cycle



4.1 Match

During the match phase of the recognize-act cycle, the run-time system compares the elements in working memory with each condition element in each production's left-hand side. The left-hand side of a production is satisfied when working-memory elements match every positive condition element and when no working-memory elements match negative condition elements. Consider the following production:

```
(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are | <VALUE> |checks dated|
                          <DAY> <MONTH> <YEAR> (CRLF)))
```

When working-memory elements match the first and third condition elements and when no working-memory elements match the second condition element, the left-hand side of this production is satisfied.

As the left-hand sides of productions are satisfied, the run-time system creates a conflict set containing records of the working-memory elements that match the condition elements of a production. Each record, called an instantiation, includes a production name and a list of the time tags of working-memory elements that match the condition elements on the production's left-hand side.

An example of an instantiation is:

```
FIND-CHECKS 12 3 11
```

FIND–CHECKS is the name of the production. The integers 12, 3, and 11 are time tags of the working-memory elements that match condition elements on the production's left-hand side. The element whose time tag is 12 matches the first condition element, the element whose time tag is 3 matches the second condition element, and the element whose time tag is 11 matches the third condition element.

More than one set of working-memory elements might satisfy the left-hand side of a production. Therefore, the conflict set might contain more than one instantiation for the same production. For example, the conflict set could contain:

```
FIND-CHECKS 12 3 11
FIND-CHECKS 12 4 11
FIND-CHECKS 12 5 11
FIND-CHECKS 12 6 11
FIND-CHECKS 12 2 11
```

4.2 Conflict Resolution

Once the conflict set is built, the recognize-act cycle proceeds to the conflict-resolution phase. During conflict resolution, the run-time system uses a strategy to select one of the instantiations in the conflict set.

Section 4.2.1 explains the rules on which the conflict-resolution strategies are based. Section 4.2.2 describes the conflict-resolution strategies.

4.2.1 Conflict-Resolution Rules

The conflict-resolution strategies are based on the following rules:

1. Refraction—Selects an instantiation only once. Refraction prevents a program from looping infinitely on the same data.
2. Recency—Selects the instantiation that refers to the most recent data in working memory. Working-memory elements that have the highest time tags contain the most recent data. Therefore, the system must select the instantiation that contains the highest time tags.
3. Specificity—Selects an instantiation of a production whose left-hand side is the most specific. Specificity is determined by the number of conditional tests on a production's left-hand side.

4.2.1.1 Refraction

Refraction prevents programs from looping infinitely on the same data by removing instantiations from the conflict set after they have been selected.

4.2.1.2 Recency

To determine which instantiation is to be selected, the run-time system determines an instantiation's recency by comparing the time tags of all instantiations in the conflict set, and selecting the most recent.

Suppose the first pair of instantiations the run-time system compares are:

```
FIND-CHECKS 12 3 11
FIND-CHECKS 12 4 11
```

The system compares the highest time tags of the instantiations. If one time tag is higher than the other, the instantiation with the higher time tag is ordered first. If the time tags are equal, the system compares the next highest time tags of the instantiations. The highest time tags for both instantiations in the example are 12. Therefore, the system compares the next highest time tags. The next highest time tags for both instantiations are 11. The system continues comparing the time tags until one time tag is higher than the other. Both instantiations in the example have one more time tag for the system to compare. Since the first instantiation contains time tag 3, and the second instantiation contains time tag 4, the second instantiation is more recent.

If one instantiation runs out of time tags before the other instantiation, the one with more time tags is ordered before the one with fewer time tags. If both instantiations have the same time tags and run out of time tags at the same time, the recency of the instantiations is equal.

4.2.1.3 Specificity

If the recency of two or more instantiations is equal, the run-time system must order the instantiations according to their specificity. An instantiation's specificity is determined by the complexity of the left-hand side of the production to which the instantiation refers.

The VAX OPS5 compiler calculates the specificity of each production in a program by counting the number of conditional tests in the production's left-hand side. The production whose left-hand side contains the most tests is the most specific. Each of the following items is considered to be a single conditional test:

- A class name
- A disjunction
- A constant value preceded by a predicate (except within a disjunction)
- An occurrence of a variable (except the first occurrence)

NOTE

The compiler does not consider an attribute to be a conditional test.

The content of a disjunction is considered to be one conditional test. For example, the following disjunction is one test:

```
<< 105 14 NOV 1988 >>
```

The compiler considers each disjunction, constant, and variable in a conjunction to be a separate test. The following conjunction contains two tests:

```
{ << 100 104 106 >> > 102 <NUMBER> }
```

Consider the following production:

```
(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are | <VALUE> |checks dated|
                          <DAY> <MONTH> <YEAR> (CRLF)))
```

The left-hand side of this production contains eight conditional tests. The first condition element contains two tests—REPLY and <> STOP. The variables <DAY>, <MONTH>, and <YEAR> are not tests, because those variables appear for the first time in the production and are bound to atoms. The second condition element contains five tests— CHECK, <DAY>, <MONTH>, <YEAR>, and NO. The variables are counted as tests in this element because the variables are bound to atoms that can be compared to the atoms in a working-memory element. The third condition element contains one conditional test—COUNT. The variable <VALUE> is not a test, because that variable appears for the first time.

4.2.2 Conflict-Resolution Strategies

The VAX OPS5 run-time system supports two conflict-resolution strategies: the lexicographic-sort (LEX) strategy and the means-ends-analysis (MEA) strategy. These strategies use the rules described in the preceding sections to order the instantiations in the conflict set.

Both strategies apply the rules in the following order: refraction, recency, specificity. However, the MEA strategy includes an extra step after refraction, which helps to organize large programs. This step orders the instantiations according to the recency of the working-memory element matching the first condition element in each production.

The default strategy for VAX OPS5 is LEX. You can change the strategy to MEA by specifying the STRATEGY command with the keyword MEA. For example:

```
OPS5>(STRATEGY MEA)
```

To change the strategy back to the default, specify the command with the keyword LEX.

```
OPS5>(STRATEGY LEX)
```

For more information about the STRATEGY command, see Chapter 12. The next two sections describe how the two strategies order the instantiations in the conflict set.

4.2.2.1 Lexicographic-Sort Strategy

The LEX strategy is for programs that do not depend on the order in which the productions are executed. The LEX strategy uses the following rules in sequence to order the instantiations in the conflict set:

1. Apply refraction by removing from the conflict set the instantiations the run-time system has selected during the previous cycle.
2. Order the rest of the instantiations according to their recency and select the instantiation with the highest level of recency.

3. If more than one instantiation has the highest level of recency, order those instantiations according to their specificity and select the instantiation with the highest level of specificity.
4. If more than one instantiation has the highest level of specificity, select an instantiation arbitrarily.

Suppose a production FIND–CHECKS contains 10 conditional tests, a production COUNTED–CHECKS contains 8 conditional tests, and the conflict set contains the following instantiations:

```
FIND-CHECKS 3 6 20
COUNTED-CHECKS 20 3 6
```

After the instantiations have been checked for refraction, the strategy checks the instantiations for recency. Since both instantiations contain the same time tags (even if they are in a different order), the instantiations are equally recent. The strategy then applies the rule of specificity. The instantiation that contains the production name FIND–CHECKS is more specific because the left-hand side of that production contains 10 conditional tests. Therefore, the first instantiation in the preceding conflict set is selected for the next phase of the recognize-act cycle.

4.2.2.2 Means-Ends-Analysis Strategy

The MEA strategy places highest priority on the production whose first condition element is matched by the most recent working-memory element. Use this strategy if you place the most important condition element first on the left-hand side of each production. The extra step of the MEA strategy checks the recency of the time tags for the working-memory elements matching these condition elements. Therefore, you can use the MEA strategy for programs that deal with problems you can divide into tasks.

The MEA strategy uses the following rules in sequence to order the instantiations in the conflict set:

1. Apply refraction by removing from the conflict set the instantiations the run-time system has selected during the previous cycle.
2. Compare the first time tag of each instantiation still remaining in the conflict set and select the instantiation with the highest level of recency.
3. If more than one instantiation has the highest level of recency for the first time tag, order those instantiations according to their recency (using all time tags) and select the instantiation with the highest level of recency.
4. If more than one instantiation has the highest level of recency, order those instantiations according to their specificity and select the instantiation with the highest level of specificity.
5. If more than one instantiation has the highest level of specificity, select an instantiation arbitrarily.

Suppose the run-time system uses the MEA strategy to select an instantiation from the conflict set in the previous section.

```
FIND-CHECKS 3 6 20
COUNTED-CHECKS 20 3 6
```

After the instantiations have been checked for refraction, the MEA strategy checks the recency of the first time tag in each instantiation. Since the first time tag for the first instantiation is 3 and the first time tag for the second is 20, the instantiation of COUNTED–CHECKS has the highest level of recency. Therefore, the run-time system selects the second instantiation for the next phase of the recognize-act cycle.

4.3 Act

After the run-time system has selected an instantiation, the recognize-act cycle enters the act phase. During this phase, variables are bound to values, and the actions on the right-hand side of the production to which the selected instantiation refers are executed. The actions are executed in the order in which they appear in the code, except for REMOVE and MODIFY actions. The run-time system executes REMOVE actions last. MODIFY becomes a MAKE and a REMOVE; the MAKE part is executed in order, the REMOVE at the end of the phase.

If a HALT action is executed, the run-time system stops executing recognize-act cycles when the current cycle is completed and returns control to the command interpreter. Otherwise, the cycle goes back to the match phase when the act phase is completed.

Part II
Writing VAX OPS5 Programs

Part II of this manual provides information on the use of the VAX OPS5 components.

Chapter 5 explains how to use the VAX OPS5 statements, actions, and functions, and provides a sample VAX OPS5 program. Chapter 6 explains how routines written in other VAX languages can be called from an OPS5 program, how an OPS5 program can be called as a subroutine from a program written in another VAX language, and how to synchronize completion routines.

Chapter 5

Using VAX OPS5 Statements, Actions, and Functions

You can use VAX OPS5 statements, actions, and functions in a VAX OPS5 program to perform the operations listed in Table 5–1 (each operation is listed with the constructs used to perform the operation). The following sections explain how to use the constructs. The last section provides a sample VAX OPS5 program that contains some of the constructs. Detailed descriptions of the statements, actions, and functions used in this chapter are provided in Chapters 9, 10, and 11, respectively.

Table 5–1: VAX OPS5 Statements, Actions, and Functions

Operation	Construct
Initialize a program	STARTUP statement
Modify working memory	MAKE action REMOVE action MODIFY action
Copy values from a working-memory element	SUBSTR function
Save and restore the state of working memory and the conflict set	SAVESTATE action ADDSTATE action RESTORESTATE action
Stop program execution	HALT action
Bind variables	BIND action LITVAL function CBIND action
Perform arithmetic computations	COMPUTE function
Perform input and output operations	OPENFILE action CLOSEFILE action DEFAULT action ACCEPT function ACCEPTLINE function WRITE action CRLF function RJUST function TABTO function
Control loops	CATCH statement AFTER action
Use system-generated atoms	GENATOM function

(continued on next page)

Table 5–1 (Cont.): VAX OPS5 Statements, Actions, and Functions

Operation	Construct
Add productions to an executing program	BUILD action
Call external subroutines	CALL action

5.1 Initializing a Program

You should initialize a VAX OPS5 program with declarations and an optional STARTUP statement. Declarations assign the fields of working-memory elements to class names and attribute names (see Section 2.3) and identify external routines. All declarations must appear before any statements that use them.

A STARTUP statement sets up initial conditions, such as the contents of working memory, the enabling or disabling of run-time messages, the conflict-resolution strategy, and the run-time system's trace level. Only one STARTUP statement can appear in a program.

A STARTUP statement can include actions and the commands @, DISABLE, ENABLE, RUN, STRATEGY, and WATCH. For descriptions of these commands, see Chapter 12.

Consider the following declarations and STARTUP statement:

```
(VECTOR-ATTRIBUTE DATE)

(LITERALIZE CHECK
            NUMBER AMOUNT COUNTED DATE)

(LITERALIZE COUNT
            VALUE)

(LITERALIZE REPLY
            DATE)
(STARTUP
     (MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
                 ^DATE 2 NOV 1988)
     (MAKE CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO
                 ^DATE 25 NOV 1988)
     (MAKE START)
     (WATCH 0)
     (DISABLE HALT)
     (STRATEGY MEA)
     (RUN))
```

The VECTOR–ATTRIBUTE declaration declares the symbol DATE to be the name of a vector attribute. Each LITERALIZE declaration associates a class name with a list of attribute names and assigns fields to those names. The STARTUP statement sets up initial conditions, and the MAKE actions create five working-memory elements.

```
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
5 [NIL] (START)
```

The WATCH command sets the run-time system's trace level to 0, the DISABLE command disables run-time informational messages and causes control to be returned to the operating system when the program halts, and the STRATEGY command sets the conflict-resolution strategy to MEA. The RUN command instructs the run-time system to start executing recognize-act cycles.

5.2 Modifying Working Memory

A VAX OPS5 program can modify the contents of working memory during execution.

5.2.1 Creating Working-Memory Elements

To create a working-memory element, use the MAKE action with a combination of the following arguments:

- A class name
- A list of scalar attributes and their values
- A vector attribute and its value

The following MAKE action creates an element whose class name is CHECK, whose scalar attributes are ^NUMBER, ^AMOUNT, and ^COUNTED, and whose vector attribute is ^DATE:

```
(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
        ^DATE 2 NOV 1988)
```

The element is stored in working memory as follows:

```
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
```

The following production contains two MAKE actions:

```
(P WHAT-DATE
     { <START>
       (START) }
   -->
     (REMOVE <START>)
     (WRITE (CRLF) (CRLF) |What date do you want to search for?|)
     (WRITE (CRLF) (CRLF) |Enter the day, the first three|
                          |letters of the month, and the year.|
                   (CRLF)
                          |For example -- 14 NOV 1988|
            (CRLF) (CRLF)
                          |Type STOP to halt the program.|
            (CRLF) (CRLF)
                          |Date>>> |)
     (MAKE COUNT ^VALUE 0)
     (MAKE REPLY ^DATE (ACCEPTLINE)))
```

The first MAKE action creates a working-memory element (which can be examined using the WM command described in Chapter 12) that has the class name COUNT and a value 0:

```
6 [WHAT-DATE] (COUNT ^VALUE 0)
```

The second MAKE action creates a working-memory element that has the class name REPLY and atoms read from the terminal by the ACCEPTLINE function. If the function reads the atoms 14, NOV, and 1988, the MAKE action creates the following working-memory element:

```
7 [WHAT-DATE] (REPLY ^DATE 14 NOV 1988)
```

For information about reading input, see Section 5.8.4.

5.2.2 Deleting Elements from Working Memory

The REMOVE action deletes elements from working memory. Specify this action with one or more element designators, which indicate the working-memory elements to be deleted. Consider the following production:

```
(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are| <VALUE> |checks dated|
                          <DAY> <MONTH> <YEAR> (CRLF)))
```

The first REMOVE action deletes the working-memory element that matches the condition element bound to the element variable <REPLY>:

```
(REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>)
```

The second REMOVE action deletes the working-memory element that matches the condition element bound to the element variable <COUNTER>:

```
(COUNT ^VALUE <VALUE>)
```

If two or more REMOVE actions contain the same argument value, the extra REMOVE actions are ignored.

The run-time system executes REMOVE actions after executing all other actions on the right-hand side, regardless of their position in the code. For example:

```
              .
              .
              .
(MAKE (SUBSTR <CHECK> DATE INF))
(REMOVE <CHECK>)
(MAKE (SUBSTR <CHECK> DATE INF))
              .
              .
              .
```

The calls to the SUBSTR function return the same value, even though the REMOVE action appears between the two calls. See Section 5.3 for information about using the SUBSTR function.

5.2.3 Changing the Atoms in Working-Memory Elements

To change atoms in a working-memory element, use the MODIFY action with an element designator, which indicates the element whose atoms you want to change, the attributes to be changed, and new atoms. The MODIFY action deletes the working-memory element indicated by the designator and then uses the attributes and their values to create a new element. The new element retains the atoms in the deleted element except for the atoms in fields of the specified attributes, which are replaced by the new atoms that you specified.

NOTE

Since you create a new working-memory element when you modify an element, the element's time tag changes.

To change the value of the ^COUNTED attribute in the working-memory element indicated by the element variable <CHECK>, specify the action:

```
(MODIFY <CHECK> ^COUNTED YES)
```

Suppose working memory contains the following elements:

```
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6 [WHAT-DATE](COUNT ^VALUE 0)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

The following production contains two MODIFY actions:

```
(P FIND-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
     { <CHECK>
       (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
              ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                          |for $| <AMOUNT>
                          |dated| (SUBSTR <REPLY> DATE INF))
     (MODIFY <CHECK>  ^COUNTED YES)
     (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The first MODIFY action modifies the working-memory element that matches the second positive condition element, by changing the value of the attribute ^COUNTED to YES.

The second MODIFY action modifies the working-memory element that matches the condition element (COUNT ^VALUE <VALUE>) by changing the value of the attribute ^VALUE to the result of the function call (COMPUTE 1 + <VALUE>).

When this production is executed, working memory changes as follows:

```
1[NIL](CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2[NIL](CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
4[NIL](CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
7[WHAT-DATE](REPLY ^DATE 14 NOV 1988)
9[FIND-CHECKS](CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED YES ^DATE 14 NOV 198
10[FIND-CHECKS] (COUNT ^VALUE 1)
```

The working-memory element whose time tag is 3 is replaced by the working-memory element whose time tag is 9, and the new value of the attribute ^COUNTED is YES. The element whose time tag is 6 is replaced by the element whose time tag is 10, and the new value of the attribute ^VALUE is 1.

5.3 Copying Atoms from a Working-Memory Element

The SUBSTR function copies a sequence of atoms from a working-memory element to output produced by the WRITE action or to another working-memory element created by the MAKE action or modified by the MODIFY action.

Specify the SUBSTR function with three arguments: an element designator and two values that mark the boundaries of the sequence of atoms the function is to copy. The SUBSTR function copies atoms from the working-memory element to which the designator refers.

The boundary markers can be attribute names, integers, or variables bound to attribute names or integers. Integers indicate specific fields in an element. The value marking the end of the sequence can also be the symbol INF, which causes the function to copy atoms until it reaches the end of the element.

The WRITE action in the following production contains a call to the SUBSTR function:

```
(P FIND-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
     { <CHECK>
       (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
             ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                          |for $| <AMOUNT>
                          |dated| (SUBSTR <REPLY> DATE INF))
     (MODIFY <CHECK>  ^COUNTED YES)
     (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The SUBSTR function copies the atoms in the value of the attribute ^DATE. The WRITE action then displays those atoms on the terminal.

5.4 Saving and Restoring the State of Working Memory and the Conflict Set

You can copy the program state, that is, the state of working memory and the conflict set, to a file by using the SAVESTATE action. The following action copies the program state to the file CHECKS.DAT:

```
(SAVESTATE CHECKS.DAT)
```

Once you have saved the program state in a file, you can use the ADDSTATE action to add the contents of that file to the current program state:

```
(ADDSTATE CHECKS.DAT)
```

If you want to clear the program state and restore it to the state produced by the SAVESTATE action, use the RESTORESTATE action. Suppose you have used the SAVESTATE action to copy the program state to the file CHECKS.DAT. You can use the following action to clear and restore the program state to that recorded in the file CHECKS.DAT:

```
(RESTORESTATE CHECKS.DAT)
```

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

The state of external user-supplied routines is not saved by the SAVESTATE action, and thus cannot be added or restored with the ADDSTATE or RESTORESTATE action.

5.5 Stopping Program Execution

You can use the HALT action to stop the run-time system from executing recognize-act cycles; program execution stops when the current recognize-act cycle ends. If informational messages are enabled, the run-time system displays the following message and invokes the command interpreter:

```
%OPSRT-I-HALTED, HALT -- right-hand-side action
```

```
OPS5>
```

If informational messages are disabled, the run-time system exits from the VAX OPS5 program. To disable and reenable informational messages, use the DISABLE and ENABLE commands described in Chapter 12.

Consider the following production, and suppose informational messages are enabled:

```
(P STOP-COUNT
     { <REPLY>
       (REPLY ^DATE STOP) }
   -->
     (REMOVE <REPLY>)
     (HALT))
```

When a working-memory element matches the condition element bound to the element variable <REPLY> and the production STOP–COUNT has the highest priority in the conflict set, the HALT action causes the run-time system to stop executing recognize-act cycles when the current cycle ends, displays the following message, and invokes the command interpreter:

```
%OPSRT-I-HALTED, HALT -- right-hand-side action
```

```
OPS5>
```

5.6 Binding Variables

You can bind variables to values by using the BIND and CBIND actions. The BIND action binds a variable to an atom. By using the BIND action with the LITVAL function, you can bind a variable to an attribute's field. The CBIND action binds an element variable to the element most recently added to working memory.

5.6.1 Binding a Variable to an Atom

Use the BIND action to bind a variable to an atom. When you specify the BIND action with a variable and a right-hand-side expression, the run-time system evaluates the expression and binds the variable to the result of the evaluation. For example:

```
(BIND <COUNTER> (COMPUTE <N> + 1))
```

The run-time system evaluates the expression (COMPUTE <N> + 1) and binds the variable <COUNTER> to the result.

You can also specify the BIND action without a right-hand-side expression:

```
(BIND <NEW-ATOM>)
```

When the action is executed, the run-time system creates a new atom and binds the specified variable <NEW–ATOM> to the atom. Each time the action is executed, a unique atom is created and bound to the variable. This is similar to the action of the GENATOM function described in Chapter 11.

5.6.2 Binding a Variable to an Attribute's Field

You can bind a variable to an attribute's field by using the LITVAL function in a BIND action. The LITVAL function returns the integer that represents an attribute's field. Specify the function with a declared attribute name or a variable bound to a declared attribute name. For example, suppose a LITERALIZE declaration has caused field 2 to be assigned to the attribute name NUMBER. The following call to the LITVAL function will return 2:

```
(LITVAL NUMBER)
```

You can also specify the LITVAL function with an integer or a variable bound to an integer. In these cases, the function returns only that integer.

The name of a vector attribute refers only to the first atom in the attribute's value. Therefore, you can refer directly only to the first atom. To refer to other atoms in the vector, you must first bind a variable to the integer representing the field of that first atom. You can then refer to subsequent fields by incrementing the integer bound to that variable.

For example, suppose the value of the vector attribute ^DATE represents the day, month, and year, and you want to change the month. If you specify the LITVAL function with the attribute name DATE, the function returns the field storing the day. To access the atom in the month field, you can specify:

```
                                   .
                                   .
                                   .
(BIND <DAY-FIELD> (LITVAL DATE))
(BIND <MONTH-FIELD> (COMPUTE <DAY-FIELD> + 1))
(MODIFY 1 ^DATE <DAY> (SUBSTR 2 <MONTH-FIELD> <MONTH-FIELD>))
                                   .
                                   .
                                   .
```

The first BIND action binds the integer that represents the field containing the first atom of the attribute's value to the variable <DAY–FIELD>. Since the month is the second atom in the value, you must add 1 to the integer bound to the variable <DAY–FIELD>, using the COMPUTE function. The second BIND action binds the new field to the variable <MONTH–FIELD>. The MODIFY action then changes the month of the working-memory element matching the first condition element in the production to the month of the working-memory element matching the second condition element.

5.6.3 Binding an Element Variable to a Working-Memory Element

The CBIND action binds an element variable to the last element added to working memory by a MAKE, MODIFY, or CALL action. For more information about element variables, see Section 3.2.2.

Consider the following actions:

```
        .
        .
        .
(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
            ^DATE 2 NOV 1988)
(CBIND <NEW-WME>)
        .
        .
        .
```

The MAKE action creates a working-memory element, and then the CBIND action binds the element variable <NEW–WME> to that element. You can then use the element variable <NEW–WME> as an element designator in a REMOVE or MODIFY action, or in a call to the SUBSTR function.

5.7 Performing Arithmetic Computations

The values you specify for many action arguments can be numbers. Rather than explicitly specifying a number, you can specify a call to the COMPUTE function. It evaluates an arithmetic expression and returns the result. An arithmetic expression can contain numbers, variables bound to numbers, and arithmetic operators. If an expression contains both an integer and a floating-point number, the result is a floating-point number.

The operators you can use are:

+	Addition
–	Subtraction
*	Multiplication
//	Division
\\	Modulus

NOTE

Use the modulus operator only with an integer or a variable bound to an integer.

Use infix notation in VAX OPS5 arithmetic expressions, that is, place operators between operands. Separate each operator and operand with a space. For example:

```
2 + <X>
```

All operators have the same priority, and the COMPUTE function evaluates them from right to left. For example, the result of the following call to the COMPUTE function is 12:

```
(COMPUTE 2 + 2 * 5)
```

To override the right-to-left evaluation, use parentheses. For example, the following call to the COMPUTE function produces 20:

```
(COMPUTE (2 + 2) * 5)
```

Consider the following production:

```
(P FIND-CHECKS
   { <REPLY>
     (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
   { <CHECK>
     (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
            ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
   { <COUNTER>
     (COUNT ^VALUE <VALUE>) }
  -->
   (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                        |for $| <AMOUNT>
                        |dated| (SUBSTR <REPLY> DATE INF))
   (MODIFY <CHECK> ^COUNTED YES)
   (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The call to the COMPUTE function calculates a new value for the attribute ^VALUE. The function adds 1 to the integer to which the variable <VALUE> is bound and returns the result as the value of the attribute ^VALUE.

NOTE

VAX OPS5 does not perform complex mathematical tasks. For complex mathematical tasks, a VAX OPS5 program can call external routines written in other VAX languages (see Chapter 6) that are optimized for algorithmic coding.

5.8 Performing Input and Output Operations

VAX OPS5 programs can read input from and write output to a terminal or a file. You can use VAX OPS5 actions to:

- Open files
- Close files
- Set the default input source and output destination
- Read input
- Write output

5.8.1 Opening Files

To open a file for reading or writing, use the OPENFILE action. Specify the action with a file identifier, a VMS file specification, and one of the keywords IN, OUT, or APPEND. If you specify IN, the action opens an existing file for reading only. If you specify OUT, the action creates a new file and opens it for writing only. If you specify APPEND, the action opens an existing file for writing and sets the file pointer to the end of the file.

After opening a file, the action associates the file identifier with that file. For example, the following action opens the file CHECKS.DAT for reading and associates the file identifier CHECKSI with the file:

```
(OPENFILE  CHECKSI CHECKS.DAT IN)
```

In the next example, the OPENFILE action opens the file CHECKS.LOG for writing, and associates the file identifier CHECKSO with the file.

```
(OPENFILE CHECKSO CHECKS.LOG OUT)
```

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Once a file has been opened and associated with a file identifier, you can use that file for input or output operations by specifying the file identifier with the following actions, functions, and commands:

- ACCEPT function (input)
- ACCEPTLINE function (input)
- CLOSEFILE action (input and output)
- CLOSEFILE command (input and output)
- DEFAULT action (input and output)
- DEFAULT command (input and output)
- WRITE action (output)

5.8.2 Setting the Default Input Source and Output Destination

Use the DEFAULT action to set the default source for input operations or the destination for output operations. The argument values you specify with the action determine the source or destination.

By default, input comes from the terminal. To set the source to a file, specify the DEFAULT action with the file identifier of an open input file and the keyword ACCEPT. Suppose CHECKSI is the file identifier for an open input file. The following action sets that file to be the default source for input:

```
(DEFAULT CHECKSI ACCEPT)
```

Once the default for input has been set to a file, all input required by the ACCEPT and ACCEPTLINE functions is taken from that file. To set the default back to the terminal, specify the symbol NIL with the keyword ACCEPT.

```
(DEFAULT NIL ACCEPT)
```

The terminal is also the default destination for output. To set the destination to a file, specify the DEFAULT action with the file identifier of an open output file and the keyword TRACE or WRITE. The keyword TRACE sets the destination for trace output (see the *VAX OPS5 User's Guide*), and the keyword WRITE sets the destination for the WRITE action. Suppose CHECKSO is the file identifier for an open output file. The following action sets that output file to be the default destination for trace output:

```
(DEFAULT CHECKSO TRACE)
```

Once the destination for trace output has been set to a file, all trace output produced by the run-time system is sent to that file. Likewise, if you substitute the keyword WRITE for TRACE, all output produced by the WRITE action is sent to that file.

```
(DEFAULT CHECKSO WRITE)
```

To set the default destination back to the terminal, specify the symbol NIL with the appropriate keyword. For example:

```
(DEFAULT NIL TRACE)
```

5.8.3 Closing Files

To close files, specify the CLOSEFILE action with the file identifiers of the files you want to close. The action closes the files associated with the file identifiers you specify and then dissociates the identifiers from the files. For example, to close files whose file identifiers are CHECKSI and CHECKSO, specify the following:

```
(CLOSEFILE CHECKSI CHECKSO)
```

Once you have closed a file, you can no longer use its file identifier with other actions, functions, or commands to perform input and output operations. To use the files again, you must reopen them.

5.8.4 Reading Input

The input functions ACCEPT and ACCEPTLINE read atoms and lists of atoms (enclosed in parentheses) from the terminal or a file. The ACCEPT function reads either an atom or a list. The ACCEPTLINE function reads a line of input consisting of one or more atoms and lists.

The MAKE action in the following production uses the ACCEPT function to read input:

```
(P WHAT-NUMBER
   { <START>
     (START) }
  -->
   (REMOVE <START> )
   (WRITE (CRLF) |What number are you looking for? |)
   (MAKE REPLY ^NUMBER (ACCEPT)))
```

The attribute ^NUMBER is given the value read by the ACCEPT function.

By default, the input functions read input from the terminal. If you want an input function to read from a file, call the function with the file identifier of an open input file or change the default for input, using the DEFAULT action (see Section 5.8.2).

When the ACCEPT function reads past the end of a file, the function returns END–OF–FILE.

You can specify the ACCEPTLINE function with default values, which can be atoms or variables bound to atoms. The function returns the default values if it reads:

- A line of input that consists of only a carriage return
- Past the end of a file

The MAKE action in the following production contains a call to the ACCEPTLINE function, which includes the default values NO and DATE:

```
(P WHAT-DATE
     { <START>
       (START) }
   -->
     (REMOVE <START>)
     (WRITE (CRLF) |What date do you want to search for? |)
     (MAKE REPLY ^DATE (ACCEPTLINE NO DATE)))
```

The function call reads the atoms you type at the terminal. The MAKE action uses the atoms to create a working-memory element that has the class name REPLY and the attribute ^DATE. If the default for input is a file, the atoms are read from that file and not from the terminal. If you press the Return key, or if the file from which the function reads does not contain an atom, the function returns the atoms NO and DATE to the attribute ^DATE.

5.8.5 Writing Output

To write output to the terminal or a file, use the WRITE action. If you want to send output to a file, specify the action with the file identifier of an open output file or change the default for the WRITE action, using the DEFAULT action (see Section 5.8.2).

The right-hand-side expression that you specify with a WRITE action is evaluated by the WRITE action, and the output is written on the current output line with one space between values.

Suppose the variable <VALUE> is bound to 5, <DAY> is bound to 2, <MONTH> is bound to NOV, and <YEAR> is bound to 1988. You could use these variables in a WRITE action as follows:

```
(WRITE There are <VALUE> checks dated <DAY> <MONTH> <YEAR>)
```

This action displays the following output on the terminal:

```
THERE ARE 5 CHECKS DATED 2 NOV 1988
```

The WRITE action writes output in uppercase characters. To display output in exactly the way you have entered it, enclose the text in vertical bars (| |). For example:

```
(WRITE |There are| <VALUE> |checks dated| <DAY> <MONTH> |<YEAR>|)
```

This action displays:

```
There are 5 checks dated 2 NOV <YEAR>
```

The variable <YEAR> rather than its value is included in the output because <YEAR> is specified between vertical bars.

You can control the format of the WRITE action's output by using the CRLF, TABTO, and RJUST functions to:

- Produce output on a new line
- Specify the column in which to start writing output
- Produce right-justified output

5.8.5.1 Producing Output on a New Line

To write output on a new line, use the CRLF function in a WRITE action. The CRLF function places an end-of-line character string (carriage-return and line-feed control characters) in the result element.

The following production contains WRITE actions that include calls to the CRLF function:

```
(P WHAT-DATE
    { <START>
      (START) }
  -->
    (REMOVE <START>)
    (WRITE (CRLF) (CRLF) |What date do you want to search for? |
           (CRLF) (CRLF) |Enter the day, the first three|
                         |letters of the month, and the year.|
                  (CRLF)
                         |For example -- 14 NOV 1988|
           (CRLF) (CRLF)
                         |Type STOP to halt the program.|
           (CRLF) (CRLF)
                         |Date>>> |)
    (MAKE COUNT ^VALUE 0)
    (MAKE REPLY ^DATE (ACCEPTLINE)))
```

The WRITE actions in this production produce the following output:

```
What date do you want to search for?

Enter the day, the first three letters of the month, and the year.
For example -- 14 NOV 1988

Type STOP to halt the program.

Date>>>
```

5.8.5.2 Specifying the Column in Which to Start Writing Output

Use the TABTO function to specify in which column the WRITE action is to start writing output. Specify the column number with an integer or a variable bound to an integer. For example:

```
(TABTO 15)
```

If the column you specify is to the left of the last column in which output has been written, the WRITE action writes the output on a new line, starting at the specified column.

The following WRITE action displays the headers of three columns:

```
(WRITE (CRLF) (TABTO 10) NUMBER
              (TABTO 25) AMOUNT
              (TABTO 40) DATE)
```

This action produces the following output:

```
NUMBER         AMOUNT         DATE
```

5.8.5.3 Producing Right-Justified Output

The WRITE action writes output right-justified in a field of a specified width if you use the RJUST function. Specify the width with an integer or a variable bound to an integer. For example:

```
(RJUST 45)
```

When a WRITE action contains the RJUST function, the WRITE action:

1. Allocates a field of the width specified, beginning at the next possible position on the output line
2. Determines the number of character positions required by the output being written
3. Inserts spaces that cause the output to be right-justified in the field

If the output being written requires more character positions than you specify for the field, the WRITE action writes the output as if the RJUST function was not specified, that is, the WRITE action inserts one space and then writes the output.

A call to the RJUST function must directly precede the value being written. You can use the RJUST function after calls to the CRLF and TABTO functions. For example:

```
(WRITE (CRLF)
       (TABTO 5)
       (RJUST 10) |250.00|)
```

The following WRITE action writes a vertical list of numbers right-justified in a column that is 10 characters wide:

```
(WRITE (CRLF) (RJUST 10) |10.06|
       (CRLF) (RJUST 10) |2.45|
       (CRLF) (RJUST 10) |56.00|
       (CRLF) (RJUST 10) |250.00|)
```

This action produces the following output:

```
     10.06
      2.45
     56.00
    250.00
```

5.9 Controlling Loops

You can control loops in a VAX OPS5 program by using a catcher. A catcher is a list of actions that are executed after a specified number of recognize-act cycles have been executed. Catchers control loops by placing a limit on the number of recognize-act cycles that can be executed before the catcher. For example, if program execution is unattended, as in a batch job, the catcher can halt the program if it does not produce results within a specified limit of recognize-act cycles.

You define a catcher with a CATCH statement, which includes a symbol and one or more actions. The symbol names the catcher, and functions as a label. A catcher's name must be unique; that is, it cannot be the same as the name of another catcher, a production, an external routine, an action, or a function that already exists in the program. When the catcher is enabled, the actions are executed.

The following CATCH statement defines a catcher named FINISH, which consists of two actions:

```
(CATCH FINISH
     (WRITE (CRLF) |Finished.|)
     (HALT))
```

The AFTER action enables a catcher by telling the run-time system when to execute the catcher. Specify the AFTER action with a positive integer and the name of the catcher you want to enable. The integer indicates the number of recognize-act cycles the run-time system is to execute before executing the specified catcher. If execution halts before the specified number of recognize-act cycles are executed, the catcher is not executed.

Only one catcher can be enabled at a time. Therefore, when you enable a catcher, you disable the catcher currently enabled.

The following action enables the catcher FINISH:

```
(AFTER 10 FINISH)
```

The run-time system executes the catcher FINISH after 10 recognize-act cycles have been executed. If program execution halts before 10 cycles have been executed, the run-time system does not execute the catcher. After a catcher has been executed, it is disabled, so this catcher will not be executed repeatedly every 10 cycles.

The following VAX OPS5 program illustrates the use of two catchers: STARTER and FINISH. An AFTER action in the STARTUP statement indicates that catcher STARTER is to be executed after one recognize-act cycle has been executed. The AFTER action in catcher STARTER enables catcher FINISH after 10 recognize-act cycles have been executed.

```
(STARTUP
     (WATCH 0)         ;Disables trace output
     (DISABLE HALT)    ;Disables halt messages
     (STRATEGY MEA)    ;Sets conflict-resolution strategy to MEA
     (MAKE START)      ;Creates working-memory element (START)
     (AFTER 1 STARTER) ;Enables catcher STARTER after 1
                       ; recognize-act cycle
     (RUN))            ;Executes recognize-act cycles

(CATCH STARTER         ;Catcher STARTER
     (WRITE (CRLF) |Counting to 10...|)
     (AFTER 10 FINISH) ;Enables catcher FINISH after the run-time
                       ; system has executed 10 productions
     (MAKE NUMBER 1))

(CATCH FINISH          ;Catcher FINISH
     (WRITE (CRLF) |Finished.|)
     (HALT))           ;Stop program

(P INITIALIZE          ;Initialize working memory
     { <START>
       (START) }
  -->
     (WRITE (CRLF) |Starting...|)
     (REMOVE <START>))

(P COUNT               ;Output numbers
     { <NUMBER>
       (NUMBER <N>) }
  -->
     (WRITE (CRLF) <N>)
     (MODIFY <NUMBER> ^2 (COMPUTE (<N> + 1))))
```

This program produces the following output:

```
Starting...
Counting to 10...
1
2
3
4
5
6
7
8
9
10
Finished.
```

5.10 Using System-Generated Atoms

The GENATOM function generates unique atoms. Each time the run-time system starts executing a program, the first occurrence of the GENATOM function generates the atom G:1. The second atom the function generates is G:2, the third atom is G:3, and so on. Because the atoms are unique within any program run, you can use them to create data structures, such as lists, to identify working-memory elements, and to associate working-memory elements.

Data structures provide programs with a mechanism to store and retrieve data. The following VAX OPS5 code shows how you can use the GENATOM function to link working-memory elements to form a list:

```
(LITERALIZE HEAD
            ID)

(LITERALIZE LIST
            ID FIRST-ELEMENT NEXT-ELEMENT)

(STARTUP (MAKE LIST ^FIRST-ELEMENT THIS)
         (MAKE LIST ^FIRST-ELEMENT THAT)
         (MAKE LIST ^FIRST-ELEMENT THESE)
         (MAKE LIST ^FIRST-ELEMENT THOSE))

(P GET-LIST-HEAD
     { <FIRST>
       (LIST) }
      -(HEAD)
   -->
     (BIND <POINTER> (GENATOM))
     (MODIFY <FIRST> ^ID <POINTER>)
     (MAKE HEAD ^ID <POINTER>))

(P MAKE-LIST
     { <UNBOUND>
       (LIST ^ID NIL) }
     { <TAIL>
       (LIST ^ID <> NIL ^NEXT-ELEMENT NIL) }
   -->
     (BIND <NEXT-ELEMENT> (GENATOM))
     (MODIFY <UNBOUND> ^ID <NEXT-ELEMENT>)
     (MODIFY <TAIL> ^NEXT-ELEMENT <NEXT-ELEMENT>))
```

The production GET–LIST–HEAD initializes the list head, which is an identification that points to the first list element. The production MAKE–LIST builds the list (THOSE THESE THAT THIS) by linking the working-memory elements that have the class name LIST. The linkage between elements is established by matching the next element (NEXT–ELEMENT) of one list with the identification (ID) of another.

Because the GENATOM function always returns a new atom, you can use the function to identify working-memory elements. For example, if a program keeps a record of transactions for a checking account, you could use the GENATOM function to assign a unique atom to each transaction. You could then identify a particular transaction by binding the atom produced by the function to a variable. The following BIND action binds the variable <TRANSACTION–ID> to the atom produced by the GENATOM function:

```
(BIND <TRANSACTION-ID> (GENATOM))
```

You can use the atoms produced by the GENATOM function to associate one working-memory element with another working-memory element. For example:

```
(P ACCOUNT-TRANSACTIONS
      .
      .
      .
   -->
     (BIND <TRANSACTION-ID> (GENATOM))
     (MAKE TRANSACTION ^TYPE <TYPE> ^ID <TRANSACTION-ID>)
     (MAKE ACCOUNT ^ID <NUMBER>
                   ^TRANSACTION-RECORD <TRANSACTION-ID>))
```

The BIND action binds the variable <TRANSACTION–ID> to the unique atom the GENATOM function creates. The first MAKE action creates a working-memory element that represents a transaction. The transaction is assigned a type and an identification name, which is the atom bound to the variable <TRANSACTION–ID>. The second MAKE action creates a working-memory element that assigns the transaction to an account. The two MAKE actions associate the elements they create by including the transaction's unique identification name as a value in both elements.

5.11 Adding Productions to an Executing Program

You can add productions to an executing program by using the BUILD action. Specify this action with a production name, a left-hand side, an arrow (– –>), and a right-hand side:

```
(BUILD production-name
       left-hand-side
     -->
       right-hand-side)
```

If the BUILD action finds an existing production with the name that you have specified, the original production is disabled and the new one is built. However, disabled productions remain in memory; therefore, if you build the same production many times, you decrease system performance.

Do not precede the left-hand side with the letter P and do not enclose the production in parentheses. The run-time system adds the P and parentheses when it creates the production.

The following production contains a BUILD action:

```
(P ADD-STOP-COUNT
   (STATUS ^READY-TO-STOP YES)
  -->
   (BUILD STOP-COUNT
       { <REPLY>
         (REPLY ^DATE STOP) }
      -->
       (REMOVE <REPLY>)
       (HALT)))
```

This production adds the following production to the executing program:

```
(P STOP-COUNT
   { <REPLY>
     (REPLY ^DATE STOP) }
  -->
   (REMOVE <REPLY>)
   (HALT))
```

By default, the BUILD action treats variables, actions, and functions as constants. If you want the BUILD action to evaluate a variable, action, or function call while the run-time system adds the production to the program, precede the variable, action, or function call with the unquote operator (\\). Suppose the variable <CITY> in the following example is bound to BOSTON.

```
(BUILD ADD-ADDRESS
     (HOUSE ^CITY \\<CITY> ^ADDRESS <ADDRESS>)
   -->
     (WRITE |New listing -- | <ADDRESS>))
```

The new production is:

```
(P ADD-ADDRESS
     (HOUSE ^CITY BOSTON ^ADDRESS <ADDRESS>)
   -->
     (WRITE |New listing -- | <ADDRESS>))
```

The unquoted variable <CITY> is evaluated to BOSTON. However, the variable <ADDRESS> is not evaluated until the production ADD–ADDRESS is executed during another recognize-act cycle. A description of the unquote operator is provided in Chapter 7.

The source code for a new production is placed in a file named OPS_$BUILD.OPS. Each time a BUILD action is executed, the run-time system creates a new version of the file. You can add the productions stored in the build files (OPS_ $BUILD.OPS;n) to a program's source code by specifying the files when you compile the program or by editing the program to include the build files.

5.12 Sample VAX OPS5 Program

```
; This VAX OPS5 program counts the number of checks that were
; written on a particular date.

(VECTOR-ATTRIBUTE DATE)

(LITERALIZE CHECK
            NUMBER AMOUNT COUNTED DATE)

(LITERALIZE COUNT
            VALUE)

(LITERALIZE REPLY
            DATE)
```

```
(STARTUP
     (STRATEGY MEA)
     (MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
                 ^DATE 2 NOV 1988)
     (MAKE CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 105 ^AMOUNT 27.25 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 106 ^AMOUNT 250.00 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 107 ^AMOUNT 16.15 ^COUNTED NO
                 ^DATE 14 NOV 1988)
     (MAKE CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO
                 ^DATE 25 NOV 1988)
     (MAKE CHECK ^NUMBER 101 ^AMOUNT 40.30 ^COUNTED NO
                 ^DATE 2 NOV 1988)
     (MAKE CHECK ^NUMBER 109 ^AMOUNT 45.80 ^COUNTED NO
                 ^DATE 30 NOV 1988)
     (DISABLE HALT)
     (MAKE START)
     (RUN))

(P WHAT-DATE
     { <START>
       (START) }
   -->
     (REMOVE <START> )
     (WRITE (CRLF) (CRLF) |What date do you want to search for?|)
     (WRITE (CRLF) (CRLF) |Enter the day, the first three|
                          |letters of the month, and the year.|
            (CRLF)
                          |For example -- 14 NOV 1988|
            (CRLF) (CRLF)
                          |Type STOP to halt the program.|
            (CRLF) (CRLF)
                          |Date>>> |)
     (MAKE COUNT ^VALUE 0)
     (MAKE REPLY ^DATE (ACCEPTLINE)))

(P FIND-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
     { <CHECK>
       (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
              ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                          |for $| <AMOUNT>
                          |dated| (SUBSTR <REPLY> DATE INF))
     (MODIFY <CHECK> ^COUNTED YES)
     (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))

(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are| <VALUE> |checks dated|
                         <DAY> <MONTH> <YEAR> (CRLF)))
```

```
(P STOP-COUNT
     { <REPLY>
       (REPLY ^DATE STOP) }
   -->
     (REMOVE <REPLY>)
     (HALT))
```

Chapter 6

Using Routines Written in Other VAX Languages

Programming tasks, such as computing mathematical expressions, manipulating strings, and editing large quantities of data, are often easier and more efficient to develop in languages other than VAX OPS5. If you are developing a VAX OPS5 program that needs to perform these types of operations, you should consider using external routines. An external routine is a function or subroutine written in a language other than VAX OPS5. VAX OPS5 programs can call external routines written in any VAX language except APL.

On the other hand, it is often useful to call a subroutine written in VAX OPS5 from a program written in another VAX language. An example might be a program controlling a real-time environment which accesses a VAX OPS5 expert system for decision support. The VAX OPS5 system supplies a number of support routines to help implement such an interface. A subroutine written in VAX OPS5 is a complete VAX OPS5 program which is executed by being called from another program. It then runs as part of the same process as the calling program.

Most VAX languages can call external routines, such as VMS and RMS system services, run-time library routines, the Database Management System (VAX DBMS), and the Common Data Dictionary (VAX CDD). If a VAX OPS5 program needs to use one of these services, the program must call an external routine written in another language that, in turn, calls that service.

The *VAX Architecture Handbook* and the *Introduction to VMS System Routines* contain detailed information about calling external routines and passing arguments. You should be familiar with these subjects before creating a VAX OPS5 program that calls external routines.

A routine that a program can call is termed a "procedure" in the manuals mentioned. This chapter, however, uses the expression "external routine" to maintain consistency with VAX OPS5 terminology.

To create a VAX OPS5 program that calls an external routine, you must perform the following steps:

1. Create the VAX OPS5 source file.
2. Create the external routine source file.
3. Compile the external routine source file into an object file.
4. Compile the VAX OPS5 source file into an object file.
5. Link the the VAX OPS5 and the external object code modules to the VAX OPS5 run-time module to produce an executable image.
6. Execute and debug the program.

You follow the same procedure when creating a program in another VAX language that calls a VAX OPS5 routine.

This chapter explains these steps, describes the result element, provides an overview of external routines, provides examples showing how VAX OPS5 programs call external routines written in other VAX languages, and shows how to handle asynchronous system traps.

6.1 Calling a VAX OPS5 Program as a Subroutine

A program written in a language other than VAX OPS5 may call a subroutine written in VAX OPS5 to perform a task that is easier to describe in VAX OPS5 than in the language used for the main program.

One way to do this is to use the LINKER to make a shared image of the VAX OPS5 program, and then link the main program and the shared image. The main program can call the VAX OPS5 program by a parameterless procedure call to the image entry point in the VAX OPS5 program.

If you use this mechanism, when the main program calls the VAX OPS5 subroutine it has very little control over its execution. In particular, it is not easy to pass parameters with the call to make the VAX OPS5 subroutine do something different each time it is called.

The best, and recommended, way to call a VAX OPS5 program as a subroutine is to:

1. Compile the VAX OPS5 program using the /NOENTRY switch.
2. Call OPS$INITIALIZE, OPS$STARTUP, OPS$RUN, and call OPS$CLEAR and OPS$CANCEL_RUN as needed, from the main program.
3. Link the main program and the VAX OPS5 object files into a single executable image.

Using this mechanism, you can call OPS$INITIALIZE to start the VAX OPS5 run-time system, call OPS$STARTUP to execute the STARTUP statement in the VAX OPS5 subroutine, and call OPS$RUN to control the execution of recognize-act cycles by the VAX OPS5 run-time system. You can pass parameters with a call by making working-memory elements at any time after OPS$INITIALIZE has been called.

You can call the VAX OPS5 subroutine more than once, by calling OPS$STARTUP and OPS$RUN as necessary. You can clear working memory between subroutine calls by calling the OPS$CLEAR support routine. The OPS$CANCEL_RUN support routine can be used, from any external routine called by the VAX OPS5 subroutine, to force the subroutine to stop executing recognize-act cycles.

NOTE

> It is not possible to use the recommended mechanism to LINK more than one VAX OPS5 program directly to a main program written in another language. This is because a single copy of the VAX OPS5 run-time system would be shared by the VAX OPS5 subroutines, causing erroneous results. It is possible to link the program in such a way that there are multiple copies of the run-time system, but it is recommended that the VAX OPS5 programs be combined and the method described above of passing parameters to the VAX OPS5 subroutine be used to specify the required functionality of the subroutine.

6.2 Result Element

The result element is a buffer the VAX OPS5 run-time system uses to interact with working memory and to store atoms used and produced by VAX OPS5 actions, functions, and support routines. The run-time system clears and places new atoms in the result element each time the system interacts with working memory.

For example, when the run-time system constructs a working-memory element, it stores the element's atoms in the result element until the element is ready to be loaded into working memory. The run-time system loads the element by copying the contents of the result element to a storage area in working memory. Likewise, when the run-time system modifies a working-memory element, the system copies the element from working memory to the result element and then modifies the contents of the result element. See Sections 6.5 and 6.6 for more details on how external routines can interact with the result element.

6.3 Overview of External Routines

External routines can be either functions or subroutines. Before you develop an external routine, you should be familiar with the differences between these two types of routines.

6.3.1 External Functions

A function usually receives arguments from a VAX OPS5 program, performs an operation (such as computing the square root of a number), and returns a value to the program.

Arguments that are passed to a function and the value a function returns must evaluate to atoms. An atom is a 32-bit longword representing a symbol or number. If a call to an external function includes arguments, the arguments are passed to the function using the immediate value mechanism by default or by reference, if required. For more information about argument-passing mechanisms, see the *VAX Architecture Handbook* or the *Introduction to VMS System Routines*.

The mechanism used to return an atom from an external routine depends on how the routine is declared in the VAX OPS5 program.

6.3.2 External Subroutines

A subroutine is usually more complex than a function. It performs a sequence of operations, and does not return a value to the VAX OPS5 program. Subroutines can perform such operations as creating working-memory elements, performing input and output operations, or calling a VAX DBMS data base.

Like calls to functions, calls to subroutines can include arguments. If a call to an external subroutine contains arguments, the VAX OPS5 run-time system creates an argument list by placing the atoms you specify in successive fields of the result element starting in field 1. The subroutine must then use a VAX OPS5 support routine to retrieve the arguments from the result element.

6.4 Declaring External Routines

Before a program can use an external routine, it must be declared using the EXTERNAL declaration statement.

An EXTERNAL declaration can take one of two forms:

- A declaration that simply specifies the routine names
- A full declaration that allows you to specify routine names, argument types, and argument-passing mechanisms

If a function is declared using the second version, it is expected to return values using the standard VMS mechanism. The result is returned in register R0.

Functions declared using the first version must use a nonstandard mechanism to return function results. The second form of the EXTERNAL declaration is therefore recommended.

The following example declares the external routines READ_NYSE_EXTRACT and SQUAREROOT using the first version mentioned above:

```
(EXTERNAL READ_NYSE_EXTRACT SQUAREROOT)
```

If declared in this way and used on the left-hand side (LHS) of a production, SQUAREROOT must return a value in R0, but if used on the right-hand side (RHS), it must return a value by calling the OPS$VALUE routine.

The following example declares the above external routines using the second version of the EXTERNAL declaration:

```
(EXTERNAL
    (READ_NYSE_EXTRACT)
    (SQUAREROOT FLOAT-ATOM (NUMERIC-ATOM BY VALUE)))
```

In this case, SQUAREROOT must always return a value in R0. It should return, but may not, an atom representing a floating-point value and take, as an argument, an atom representing either a floating-point number or an integer.

The type specifications are optional and may be checked at compile time or run time if supplied. The mechanism specification for the argument is optional. If the mechanism is not specified, the default is to pass an argument by value.

6.5 Calling External Routines

The methods for calling functions and subroutines are different. To call a function, you can include the function's name and arguments in a condition element on the left-hand side of a production or as an argument in an action on the right-hand side of a production. To call a subroutine, use the CALL action or command.

6.5.1 Calling Functions

You can call a function from either the left-hand or the right-hand side of a production. However, functions called from condition elements should not make, remove, or modify elements in working memory. The format for calling a function is the same as the format for calling the functions defined in VAX OPS5. A function call must include the function's name and can optionally include a list of arguments. The function name you specify must have been previously declared with the EXTERNAL declaration.

Arguments must evaluate to atoms. If an argument is a variable, it must be bound to an atom in a condition element on the left-hand side or by a BIND action on the right-hand side prior to the call. The following call to the function SQUAREROOT uses the variable <VARIANCE> as an argument:

```
(SQUAREROOT <VARIANCE>)
```

To call a function from the left-hand side of a production, place the function call in a condition element. For example, the following condition element calls the function SQUAREROOT:

```
(MEANSD ^MEAN <MEAN> ^STDDEVIATION (SQUAREROOT <VARIANCE>))
```

When the function call executes, the atom bound to the variable <VARIANCE> is passed to the function SQUAREROOT. The function SQUAREROOT then calculates the square root of that value and places the result in the result element. The atom the SQUAREROOT function returns is assigned to the attribute ^STDDEVIATION and is then used by the program to produce matches with working-memory elements.

To call a function from the right-hand side, specify the function call as an argument in an action. For example, the following BIND action contains a call to the function SQUAREROOT:

```
(BIND <STDDEVIATION> (SQUAREROOT <VARIANCE>))
```

The atom bound to the variable <VARIANCE> is passed by value to the function SQUAREROOT. The function SQUAREROOT then calculates the square root of that value and the result is bound to the variable by the BIND action.

6.5.2 Calling Subroutines

You can call a subroutine with the CALL action or command. You must specify the name of the subroutine and can optionally include a list of arguments. The subroutine you specify must have been previously declared with the EXTERNAL declaration.

The arguments in a CALL action must evaluate to atoms. Argument values can be symbols, numbers, variables, or function calls. If an argument is a variable, it must be bound to an atom in a condition element on the left-hand side or by a BIND action on the right-hand side prior to the call. The VAX OPS5 run-time system places the argument values in successive fields of the result element, starting with the first field.

The following CALL action calls a subroutine named READ_NYSE_EXTRACT with the variable <NAME> as an argument:

```
(CALL READ_NYSE_EXTRACT <NAME>)
```

The run-time system places the atom bound to the variable <NAME> in the first field of the result element. The subroutine READ_NYSE_EXTRACT then retrieves the atom, using a VAX OPS5 support routine. After the subroutine completes its execution, the run-time system executes the next action on the production's right-hand side.

You can also call external subroutines with the CALL command at the command interpreter level. Use the CALL command for debugging by making sure the subroutine correctly creates working-memory elements.

6.6 Creating the External Routine

Before developing an external routine, you should know the routine's objective, the language in which you are going to develop the routine, whether the routine is going to be a function or a subroutine, what arguments and passing mechanisms you are going to specify with the routine, and the value the routine will return (if it is a function). This information determines:

- Whether you need to declare the arguments in a function's definition
- What VAX OPS5 support routines the external routine must declare
- What VAX OPS5 support routines the external routine needs to use

The VAX OPS5 run-time system provides support routines that you can include in an external routine. Each support routine you use must be declared. Once the support routines are declared, the external routine can call them to perform operations, such as converting data types, creating working-memory elements, or displaying warning messages.

The following sections explain how to declare arguments in function definitions and how to declare and use VAX OPS5 support routines. The VAX BASIC examples use the convention that variable names represent floating-point variables unless they end with a percent sign (%) or dollar sign ($). A percent sign indicates an integer variable and a dollar sign indicates a character string. For example, ATOM% is a 32-bit longword that represents a VAX OPS5 integer atom.

6.6.1 Declaring Arguments for Functions

An external function must declare the arguments that are passed to it. The declaration must include the same number of arguments that you specify in the function call and must define each argument as a longword.

Suppose a VAX OPS5 program contains a BIND action that binds the variable <STDDEVIATION> to the result of a function call.

```
(BIND <STDDEVIATION> (SQUAREROOT <VARIANCE>))
```

The argument (SQUAREROOT <VARIANCE>) is a call to the function SQUAREROOT. The function call contains one argument—the variable <VARIANCE>. Therefore, the function SQUAREROOT must declare that argument. For example:

```
100     FUNCTION LONG SQUAREROOT (LONG ATOM%)
```

This line of code defines the function name SQUAREROOT and declares both the SQUAREROOT function and the argument to be passed to that function. The expression (LONG ATOM%) declares the argument as a longword whose name is ATOM%.

6.6.2 Declaring VAX OPS5 Support Routines

You must declare each support routine as an external routine call. This applies equally to functions and subroutines. Use the appropriate declaration for the language you are using. The VAX OPS5 compiler kit contains a number of %INCLUDE files containing all the necessary definitions for each VAX language supported by VAX OPS5 calling mechanisms.

For example, if you are interfacing a VAX PASCAL program to VAX OPS5, you would %INCLUDE the OPSDEF.PAS file in your program as follows:

```
{ Declarations }
    .
    .
    .
{ Include VAX OPS5 routine declarations }
%INCLUDE 'OPS$LIBRARY:OPSDEF.PAS'

{ Procedure DoSomething }
    .
    .
    .
```

For a VAX BASIC external routine to access the OPSDEF.BAS declarations, place the following %INCLUDE statement in the routine:

```
%INCLUDE "OPSDEF.BAS"
```

Table 6–1 lists the languages and INCLUDE files supported by the VAX OPS5 compiler kit:

Table 6–1: Include Files

Language	File
VAX Ada	OPS$LIBRARY:OPSDEF.ADA
VAX BASIC	OPS$LIBRARY:OPSDEF.BAS
VAX C	OPS$LIBRARY:OPSDEF.H
VAX FORTRAN	OPS$LIBRARY:OPSDEF.FOR
VAX PASCAL	OPS$LIBRARY:OPSDEF.PAS
BLISS–32	OPS$LIBRARY:BLI32OPS5.REQ

6.6.3 Using VAX OPS5 Support Routines

Arguments for support routines can be optional or required. Enclose all the arguments in parentheses and separate them with a comma (,) and a space. Ensure that all arguments are of the correct types and that you are using the correct mechanisms. If a support routine does not require any arguments, use empty parentheses, except when you are using the VAX Ada language, as follows:

```
OPS$WRITE ()
```

Use the support routines in an external routine to:

- Retrieve arguments from the result element (external subroutines only)
- Convert data types
- Compare atoms for equality
- Place atoms in the result element
- Create working-memory elements
- Stop program execution
- Display warning messages
- Use files opened by the VAX OPS5 program
- Read input from a terminal or file
- Write output to a terminal or file

- Generate atoms

The following sections explain how to use the support routines to perform these operations. Detailed descriptions of the routines are provided in Chapter 13.

6.6.4 Creating Working-Memory Elements

An external subroutine can create a working-memory element if it:

1. **Clears the result element**. Use the OPS$RESET routine to delete atoms in the result element.
2. **Performs necessary data conversions**. If necessary, use the OPS$INTERN, OPS$CVNA, or OPS$CVFA routine to convert each value being placed in the result element to an atom.
3. **Places atoms in the result element**. Use the OPS$VALUE routine to place each atom in the result element. If you want to place an atom in a particular field of the result element, use the OPS$TAB routine.
4. **Copies the contents of the result element to working memory**. After all the atoms are placed in the result element, use the OPS$ASSERT routine to copy the contents of the result element to working memory.

The following VAX BASIC code creates a working-memory element:

```
              .
              .
              .
CALL OPS$RESET ()
WME_CLASS% = OPS$INTERN
                (CLASS$ BY REF, LEN (CLASS$) BY VALUE)
CALL OPS$VALUE BY VALUE (WME_CLASS%)
ATOM% = OPS$INTERN (TOTAL_SHARES$ BY REF, 6% BY VALUE)
CALL OPS$VALUE BY VALUE (ATOM%)
ATOM% = OPS$INTERN (HI_PRICE$ BY REF, 6% BY VALUE)
CALL OPS$VALUE BY VALUE (ATOM%)
ATOM% = OPS$INTERN (LOW_PRICE$ BY REF, 6% BY VALUE)
CALL OPS$VALUE BY VALUE (ATOM%)
ATOM% = OPS$INTERN (CLOSING_PRICE$ BY REF, 6% BY VALUE)
CALL OPS$VALUE BY VALUE (ATOM%)
CALL OPS$ASSERT ()
              .
              .
              .
```

The call to the OPS$RESET routine clears the result element. The OPS$INTERN routine converts the character strings to atoms. After each conversion, the atom is assigned to the variable ATOM%, and the OPS$VALUE routine places the atom in the result element. The OPS$ASSERT routine then copies the contents of the result element to working memory.

If you want to retrieve the integer atom associated with an attribute name, use the OPS$LITBIND or OPS$LITVAL routine. Specify the OPS$LITBIND routine with the name of an attribute. If a field has been previously assigned to that attribute with a LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration, the support routine returns the integer that represents that field. For example, suppose a LITERALIZE declaration assigned field 2 to the attribute name NUMBER. The following call to the OPS$LITBIND routine will return the integer atom 2:

```
          .
          .
          .
NUMBER$ = "NUMBER"
ATOM% = OPS$INTERN (NUMBER$ BY REF, 6% BY VALUE)
NFIELD% = OPS$LITBIND (ATOM%)
          .
          .
          .
```

If you specify the name of an attribute that is not declared in the VAX OPS5 program, the support routine returns the attribute's name.

An external subroutine can place the integer atom representing an attribute's field in the result element by using the OPS$LITVAL routine. Specify the routine with the name of an attribute. The routine calls the OPS$LITBIND routine to retrieve the integer atom associated with the attribute name and then calls the OPS$VALUE routine to place that atom in the result element. Thus, you do not have to use the OPS$VALUE routine explicitly to return the atom to the result element.

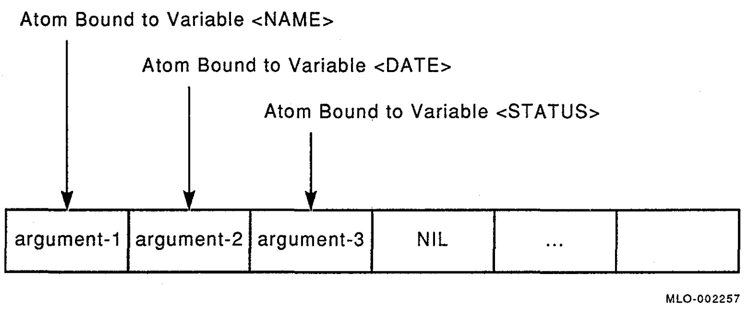
6.6.5 Retrieving Arguments from the Result Element

When a call to an external subroutine contains arguments, the run-time system places the argument values in the result element, starting in field 1. Consider the following CALL action:

```
(CALL READ_NYSE_EXTRACT <NAME> <DATE> <STATUS>)
```

This call to the subroutine READ_NYSE_EXTRACT includes three arguments: <NAME>, <DATE>, and <STATUS>. Figure 6–1 illustrates how the run-time system places the values of these arguments in the result element.

Figure 6–1: Arguments Stored in the Result Element


Atom Bound to Variable <NAME>
Atom Bound to Variable <DATE>
Atom Bound to Variable <STATUS>
argument-1
argument-2
argument-3
NIL
...
MLO-002257

To retrieve argument values from the result element, an external subroutine must include the OPS$PARAMETERCOUNT and OPS$PARAMETER support routines. The OPS$PARAMETERCOUNT routine returns to the subroutine an integer that indicates the number of argument values stored in the result element.

The OPS$PARAMETER routine retrieves an argument value stored in the result element. Specify this routine with an integer that indicates the field from which you want an argument value to be retrieved. You must specify this support routine for each argument. That is, if a CALL action contains three arguments, you must specify the OPS$PARAMETER routine three times. For example:

```
                  .
                  .
                  .
        IF OPS$PARAMETERCOUNT () <> 3% THEN
                  .
                  .
                  .
2000    NAME% = OPS$PARAMETER (1% BY VALUE)
                  .
                  .
                  .
        DATE% = OPS$PARAMETER (2% BY VALUE)
                  .
                  .
                  .
        STATUS% = OPS$PARAMETER (3% BY VALUE)
                  .
                  .
                  .
```

The OPS$PARAMETERCOUNT routine checks the number of argument values that are stored in the result element. If, for example, the support routine returns a value that is not equal to three, the subroutine displays a warning message. The OPS$PARAMETER routine retrieves the three argument values and assigns them to the variables NAME%, DATE%, and STATUS%.

6.6.6 Converting Data Types

In a VAX OPS5 program, data can be represented only by symbolic and numeric atoms. Therefore, external routines usually need to convert an argument value or a return value to a different data type. Using support routines, an external routine can:

- Translate a symbolic atom to a character string or a character string to a symbolic atom
- Convert an integer atom to an integer or an integer to an integer atom
- Convert a floating-point atom to a floating-point number or a floating-point number to a floating-point atom

The type of conversion depends on whether the argument's value is a symbolic, integer, or floating-point atom.

6.6.6.1 Symbolic Atoms

To check whether an argument value is a symbolic atom, use the OPS$SYMBOL routine. Specify the routine with an atom. If the atom is a symbol, the support routine returns 1. Otherwise the routine returns 0. For example, suppose you specify the OPS$SYMBOL routine with the atom STKNAME%:

```
           .
           .
           .
SYMBOL_TEST% = OPS$SYMBOL (STKNAME%)
           .
           .
           .
```

If the atom is symbolic, 1 is assigned to the variable SYMBOL_TEST%; otherwise 0 is assigned to the variable.

An external routine can translate a symbolic atom to a character string by using the support routine OPS$PNAME. The routine translates the atom, copies the character string to a user-supplied buffer, and returns an integer that represents the number of characters in the string. Specify the OPS$PNAME routine with three arguments; the symbolic atom to be translated, the address of the buffer to which the string is to be copied, and an integer that represents the number of characters in the string (string's length). You can use the following VAX BASIC statement to translate the symbolic atom STKNAME% and store the character string at the address STOCK_NAME$:

```
           .
           .
           .
CHARS% = OPS$PNAME(STKNAME%, STOCK_NAME$ BY REF, 6%)
           .
           .
           .
```

The integer that represents the number of characters in the string is assigned to the variable CHARS%.

If the value you specify for the length argument is less than the number of characters in the string being copied, the support routine copies only the number of characters indicated by the value of the argument.

NOTE

> The OPS$PNAME routine returns the integer representing the number of characters in the character string, even if the routine does not copy the entire string.

To convert a character string to a symbolic atom, use the OPS$INTERN support routine. Specify this support routine with two arguments: the address and the length of the character string to be translated. The support routine returns the symbolic atom produced from the translation. For example, if you want to translate the character string whose address is TOTAL_SHARES$ and whose length is six characters, specify:

```
           .
           .
           .
ATOM% = OPS$INTERN (TOTAL_SHARES$ BY REF, 6% BY VALUE)
           .
           .
           .
```

The symbolic atom the support routine returns is assigned to the variable ATOM%.

6.6.6.2 Integer Atoms

To check whether an argument value is an integer atom, use the OPS$INTEGER routine. Specify the routine with an atom. If the atom is an integer, the support routine returns 1. Otherwise the routine returns 0.

An external routine can convert an integer atom to an integer by using the support routine OPS$CVAN. Specify the routine with the atom that is to be converted. Suppose you want to test whether the value of the variable ATOM% is an integer atom, and if the result is true, you want to convert the atom to an integer.

```
          .
          .
          .
If OPS$INTEGER (ATOM%)
     THEN INTEGER% = OPS$CVAN (ATOM%)
          .
          .
          .
```

If the value of the variable ATOM% is true, the OPS$CVAN routine converts the integer atom to an integer and the result of the conversion is assigned to the variable INTEGER%.

Use the OPS$CVNA support routine to convert an integer to an atom. Specify the routine with the integer to be converted. Suppose the value of the variable INTEGER% is an integer. To convert the integer to an atom, specify:

```
          .
          .
          .
ATOM% = OPS$CVNA (INTEGER% BY VALUE)
          .
          .
          .
```

The result of the conversion is assigned to the variable ATOM%.

6.6.6.3 Floating-Point Atoms

To check whether an argument value is a floating-point atom, use the OPS$FLOATING routine. Specify the routine with an atom. If the atom is a floating-point number, the support routine returns 1. Otherwise the routine returns 0.

An external routine can convert a floating-point atom to a floating-point number by using the support routine OPS$CVAF. Specify the routine with the atom that is to be converted. Suppose you want to test whether the value of the variable ATOM% is a floating-point atom, and if the result is true, you want to convert the atom to a floating-point number.

```
          .
          .
          .
If OPS$FLOATING (ATOM%)
     THEN FLOATING_POINT = OPS$CVAF (ATOM%)
          .
          .
          .
```

If the value of the variable ATOM% is true, the OPS$CVAF routine converts the atom to a floating-point number and the result of the conversion is assigned to the variable FLOATING_POINT.

Use the OPS$CVFA support routine to convert a floating-point number to an atom. Specify the routine with the number to be converted. Suppose the value of the variable FLOATING_POINT% is a floating-point number. To convert the number to an atom, you can specify:

```
          .
          .
          .
ATOM% = OPS$CVFA (FLOATING_POINT% BY VALUE)
          .
          .
          .
```

The result of the conversion is assigned to the variable ATOM%.

6.6.7 Comparing Atoms for Equality

An external routine can compare two atoms by using the OPS$EQL support routine. The result of an atom comparison is true if one of the following statements is true:

- The atoms are both symbols that consist of the same characters.
- The atoms are the same integer.
- The atoms are the same floating-point number.

If the result is true, the support routine returns 1. If the result is false, the routine returns 0. In the following example, the OPS$EQL routine returns 1.

```
          .
          .
          .
TEST% = OPS$EQL (OPS$CVNA (2), OPS$CVNA (2))
          .
          .
          .
```

6.6.8 Placing Atoms in the Result Element

External subroutines can use the OPS$VALUE support routine to place atoms in the result element. Functions use this support routine to return an atom to a VAX OPS5 program. Subroutines use the routine to create working-memory elements.

The following code shows how a function uses the OPS$VALUE routine to return the result of a calculation to the main program:

```
          .
          .
          .
ATOM% = OPS$CVNA (INTEGER% BY VALUE)
CALL OPS$VALUE BY VALUE (ATOM%)
          .
          .
          .
```

The OPS$CVNA routine converts the result to an atom. The OPS$VALUE routine then places the atom in the result element.

Use the OPS$VALUE routine in subroutines to create working-memory elements.

The OPS$RESET routine deletes the atoms in the result element. Therefore, when a call to the OPS$VALUE routine follows a call to the OPS$RESET routine, OPS$VALUE places an atom in the result element's first field. For example:

```
          .
          .
          .
CALL OPS$RESET ()
CALL OPS$VALUE BY VALUE (WME_CLASS%)
          .
          .
          .
```

You can specify the field of the result element in which the next atom entry is to be placed by using the OPS$TAB routine. You can indicate the field with an integer or the name of an attribute that was declared in the VAX OPS5 program in a LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration. For example, if you want an atom to be placed in the third field of the result element, you can call the OPS$TAB and OPS$VALUE routines as follows:

```
          .
          .
          .
CALL OPS$TAB BY VALUE (OPS$CVNA (3% BY VALUE))
CALL OPS$VALUE BY VALUE (ATOM%)
          .
          .
          .
```

6.6.9 Stopping Program Execution

The OPS$HALT support routine causes the run-time system to stop executing a program after completing the current recognize-act cycle. If halt messages are enabled, the run-time system passes control to the VAX OPS5 command interpreter; otherwise, the system passes control to the operating system, or the program calling the VAX OPS5 routine. You can enable and disable halt messages, using the VAX OPS5 commands ENABLE and DISABLE (see the *VAX OPS5 User's Guide*).

6.6.10 Displaying Warning Messages

If you want an external routine to display warning messages on the terminal at a particular point during a routine's execution, specify the OPS$WARNING routine with:

- The address of the first byte of the character string in which the warning message is stored
- The length of the character string that stores the warning message
- One or more atoms the routine is to display in front of the warning message. These atoms are optional.

Consider the following VAX BASIC example:

```
          .
          .
          .
IF OPS$PARAMETERCOUNT () <> 1%
     THEN Z$ = ("Enter only the name of a stock") &
      \ CALL OPS$WARNING (Z$ BY REF, LEN (Z$) BY VALUE) &
      \ EXIT SUB
          .
          .
          .
```

The OPS$PARAMETERCOUNT routine returns the number of argument values that are in the result element. If the argument count is 1, the external routine continues to execute. If the argument count is an integer greater than 1, the external routine calls the OPS$WARNING routine to display the following warning message and exits:

```
Enter only the name of a stock
```

6.6.11 Using Files Opened by the VAX OPS5 Program

If an external routine needs to read from or write to a file opened by the VAX OPS5 program, the external routine must include the OPS$IFILE or OPS$OFILE routine. The OPS$IFILE routine returns the address of an input file's record access block (RAB). The OPS$OFILE routine returns the address of an output file's RAB. For more information about RABs, see the *VMS Record Management Services Manual.*

Specify the OPS$IFILE and OPS$OFILE routines with the atom representing the file identifier of a file opened in the VAX OPS5 program. Specify the name of an open input file if you use the OPS$IFILE routine, or the name of an open output file if you use the OPS$OFILE routine. For example, suppose a VAX OPS5 program opened a file for input and assigned that file the file identifier STOCK_DATA. If the same program calls an external routine, the routine can use the open input file if the routine includes the OPS$IFILE routine. A call to the routine might look like the following:

```
          .
          .
          .
FILEID$ = "STOCK_DATA"
FILEID_ATOM% = OPS$INTERN
       (FILEID$ BY REF, LEN(FILEID$) BY VALUE)
RAB% = OPS$IFILE (FILEID_ATOM%)
          .
          .
          .
```

The RAB of the file associated with STOCK_DATA is assigned to the variable RAB%.

If you specify the OPS$IFILE or OPS$OFILE routine with a file identification name that is not associated with an open input or open output file, the routine returns 0.

6.6.12 Reading Input

An external routine can read input from the terminal or a file and place the input in the result element by using the OPS$ACCEPT or OPS$ACCEPTLINE support routine. By default, these support routines read input from the terminal. If you want an external routine to read input from a file, specify the support routine with a file identifier or set a file as the default source with the DEFAULT action, prior to calling the external routine.

Assume the file identifier STOCK_DATA is associated with the open input file STOCK.DAT for the following example:

```
          .
          .
          .
FILEID$ = "STOCK_DATA"
FILEID_ATOM% = OPS$INTERN
       (FILEID$ BY REF, LEN(FILEID$) BY VALUE)
CALL OPS$ACCEPT BY VALUE (FILEID_ATOM%)
          .
          .
          .
```

The OPS$ACCEPT routine reads input from the file STOCK.DAT and places the input in the result element.

The OPS$ACCEPT routine reads an atom or a list of atoms. OPS$ACCEPT also removes the outermost parentheses from a list and matches, or balances, parentheses when reading the list.

The OPS$ACCEPTLINE routine reads a line of input that consists of atoms and lists. OPS$ACCEPTLINE simply removes all unquoted parentheses without matching.

To determine whether the input is an atom or a list, the support routines check the first printing character in the input. The routine assumes that any first character other than a parenthesis indicates an atom and places the atom in the result element. In addition to a file identifier, you can specify the OPS$ACCEPTLINE routine with default values, which can be atoms or variables bound to atoms. The routine places the default values in the result element when the routine reads:

- A line of input from the terminal that consists of only a carriage return
- A file that consists of a line of only spaces and tabs
- Past the end of a file

Suppose a VAX BASIC external routine contains the following code:

```
                         .
                         .
                         .
STOCK_INPUT$ = "STOCK_INPUT"
STOCK_INPUT% = OPS$INTERN
                    (STOCK_INPUT$ BY REF, 11% BY VALUE)
DATE$ = "12-OCT-1988"
DATE% = OPS$INTERN (DATE$ BY REF, 11% BY VALUE)
NAME$ = "DISNEY"
NAME% = OPS$INTERN (NAME$ BY REF, 6% BY VALUE)
                         .
                         .
                         .
CALL OPS$ACCEPTLINE BY VALUE (STOCK_INPUT%, DATE%,
                                            NAME%)
                         .
                         .
                         .
```

If the value of the variable STOCK_INPUT% is a symbolic atom representing a file identifier, the OPS$ACCEPTLINE routine reads input from the input file. Otherwise, the routine reads the input that is entered at the terminal. If you supply the routine with user input, the routine places that input in the result element and disregards the atoms STOCK_INPUT, 12–OCT–1988, and DISNEY. If you just press the Return key, the routine places the atoms STOCK_INPUT, 12–OCT–1988, and DISNEY in the result element.

If the OPS$ACCEPT reads past the end of a file, or if the OPS$ACCEPTLINE routine reads past the end of a file and default values are not supplied, the routines place the symbol END–OF–FILE in the result element.

Using the OPS$ACCEPT and OPS$ACCEPTLINE routines is similar to using the VAX OPS5 ACCEPT and ACCEPTLINE functions described in Chapter 11.

6.6.13 Writing Output

An external routine can display the atoms in the result element on the terminal or write the atoms to a file by using the OPS$WRITE routine. This support routine is useful for debugging because it lets you preview the contents of the result element when the external routine executes.

The atom stored in the first field of the result element determines whether the atoms are displayed on the terminal or are written to a file. If the first field contains the file identifier of a file opened for output by the VAX OPS5 program, the routine writes the remaining atoms to the associated file. Otherwise, the atoms in the result element are written to the default output file.

Suppose the symbol STOCK_DATA is a file identifier of an open output file and the result element contains the following values:

```
(STOCK_DATA STOCK_EXTRACT 100 125.25 67 105.125)
```

When the following code executes, the OPS$WRITE routine writes the atoms STOCK_EXTRACT, 100, 125.25, 67, and 105.125 to the file associated with the file identifier STOCK_DATA:

```
          .
          .
          .
CALL OPS$WRITE ()
          .
          .
          .
```

If the atom in the first field is not a file identifier, the routine writes the atoms in the result element to the default destination, which is the terminal unless you change the destination with the VAX OPS5 DEFAULT action or command.

If you want the OPS$WRITE routine to write output on more than one line, precede the call to the OPS$WRITE routine with calls to the OPS$CRLF routine. The OPS$CRLF routine places an end-of-line character string, which consists of a carriage-return character and a line-feed character, in the current field of the result element.

Using the OPS$WRITE and OPS$CRLF routines is similar to using the WRITE action and the CRLF function described in Chapter 11.

6.6.14 Generating Atoms

An external routine can generate and return new atoms by using the OPS$ATOM routine. Each atom the run-time system generates is unique. Each time the run-time system starts executing a program, the first occurrence of the OPS$ATOM support routine generates the atom G:1. The second atom the routine generates is G:2, the third atom is G:3, and so on.

An external routine can place the new atom in the result element, using the OPS$VALUE routine. For example:

```
          .
          .
          .
NEW_ATOM% = OPS$ATOM ()
CALL OPS$VALUE BY VALUE (NEW_ATOM%)
          .
          .
          .
```

The OPS$GENATOM routine calls the OPS$ATOM routine to create the new atom and then calls the OPS$VALUE routine to place the atom in the result element. Thus, you do not have to use the OPS$VALUE routine explicitly to return the value to the result element.

6.7 Examples of Calling External Routines

The following sections provide examples of VAX OPS5 programs that call external routines. Section 6.7.1 shows a program that calls an external function, and Section 6.7.2 shows a program that calls an external subroutine. Section 6.7.3 provides a program that calls an external routine that calls a VAX DBMS data base.

6.7.1 A VAX OPS5 Program That Calls an External Function

This section contains a VAX OPS5 program that computes and displays information about the mean and standard deviation of two numbers. The program calls an external function that calculates and returns the square root of a number. Examples of the function's source code and the include file are provided in VAX BASIC, VAX FORTRAN, and VAX PASCAL. For examples of INCLUDE files to use with other languages, look at the INCLUDE files supplied with your VAX OPS5 compiler.

6.7.1.1 VAX OPS5 Program—STATISTICS.OPS

```
; STATISTICS.OPS

; Declare the call-out function
(EXTERNAL (SQUARE_ROOT FLOAT-ATOM (NUMERIC-ATOM BY REFERENCE)))
; SQUARE_ROOT calculates the square root of a number and returns a
; floating-point number

(LITERALIZE ELEMENT
     NUMBER                ; Data entered at terminal
     COUNT                 ; Numbers entered
     SUM                   ; Sum of the numbers entered
     SUM_SQ)               ; Sum of the squares of the numbers

(LITERALIZE MEANSD
     MEAN                  ; Average of the numbers
     STD_DEVIATION)        ; "Spread" of the numbers

(STARTUP
     (MAKE START)
     (WATCH 2)
     (RUN))

(P INIT
     {<GO> (START)}
   -->
     (WRITE (CRLF) |This program calculates statistics for a |
                   |sequence of numbers.|
            (CRLF) |Enter a number: |)
     (MAKE ELEMENT ^NUMBER (ACCEPT) ^COUNT 0 ^SUM 0.0 ^SUM_SQ 0.0)
     (REMOVE <GO>))
```

```
(P GET-NUMBERS
     { <ELEM> (ELEMENT ^NUMBER { <NUMBER> <> STOP } ^SUM <SUM>
                       ^COUNT <COUNT> ^SUM_SQ <SUM_SQ>) }
   -->
     (WRITE (CRLF) |Enter another number or STOP to exit: |)
     (MODIFY <ELEM> ^NUMBER (ACCEPT) ^COUNT (COMPUTE <COUNT> + 1)
                    ^SUM (COMPUTE <SUM> + <NUMBER>)
                    ^SUM_SQ (COMPUTE <SUM_SQ> + <NUMBER> * <NUMBER>)))

(P CALCULATE-STATISTICS
     (ELEMENT ^NUMBER STOP ^SUM <SUM> ^COUNT <COUNT> ^SUM_SQ <SUM_SQ>)
   -->
     (WRITE (CRLF) |Sum of | <COUNT> | numbers is | <SUM>
            (CRLF) |Sum of the squares of the numbers is | <SUM_SQ>)
     (WRITE (CRLF) |Computing mean and standard deviation ...|)
     (BIND <MEAN> (COMPUTE <SUM> // <COUNT>))
     (BIND <VARIANCE> (COMPUTE (<SUM_SQ> - (<COUNT> *
                               <MEAN> * <MEAN>)) // (<COUNT> - 1)))

        ; Pass the <VARIANCE> as an argument to the external function
        ; The return value in the result element will be bound to
        ; <STD_DEVIATION>

     (BIND <STD_DEVIATION> (SQUARE_ROOT <VARIANCE>))
     (WRITE (CRLF) |Mean is | <MEAN>)
     (WRITE |, Standard deviation is| <STD_DEVIATION>)
     (HALT))
```

6.7.1.2 VAX BASIC External Function—FNSQRT.BAS

```
!FNSQRT.BAS

FUNCTION LONG SQUARE_ROOT (LONG ATOM)
! Definitions of VAX OPS5 support routines
%INCLUDE "OPS$LIBRARY:OPSDEF.BAS"
DECLARE SINGLE R

! Convert a VAX OPS5 atom to a floating-point number
IF OPS$FLOATING (ATOM BY VALUE) THEN
   R = OPS$CVAF (ATOM BY VALUE)
ELSE
! Convert a VAX OPS5 atom to an integer
   IF OPS$INTEGER  (ATOM BY VALUE) THEN
      R = OPS$CVAN (ATOM BY VALUE)
   END IF
END IF

! Convert a floating-point number to a VAX OPS5 atom
SQUARE_ROOT = OPS$CVFA (SQR(R) BY VALUE)

END FUNCTION
```

6.7.1.3 VAX FORTRAN External Function—FNSQRT.FOR

```
C       FNSQRT.FOR

        INTEGER FUNCTION SQUARE_ROOT (ATOM)

C       Definitions of VAX OPS5 support routines
        INCLUDE 'OPS$LIBRARY:OPSDEF.FOR'

        INTEGER ATOM
        REAL*4  R

C       Convert a VAX OPS5 atom to an integer or a floating-point number
C       Built-in function, %VAL, forces immediate-value mechanism
        IF (OPS$INTEGER(%VAL (ATOM))) THEN

C       Convert a VAX OPS5 atom to an integer
           R = OPS$CVAN (%VAL (ATOM))
        ELSE IF (OPS$FLOATING (%VAL (ATOM))) THEN
```

```
C      Convert a VAX OPS5 atom to a floating-point number
          R = OPS$CVAF (%VAL (ATOM))
       END IF

       SQUARE_ROOT = OPS$CVFA (%VAL (SQRT(R)))

       END
```

6.7.1.4 VAX PASCAL External Function—FNSQRT.PAS

```
(*FNSQR.PAS*)

module sqr;  (* Call out from VAX OPS5 V3.0 to VAX-11 PASCAL V3.0 *)

(* Definitions of VAX OPS5 support routines *)

%include 'ops$library:opsdef.pas'

[global] function square_root (atom: integer):integer;

var
   r : real;

begin

(* Convert a VAX OPS5 atom to a floating-point number or an integer. *)

if ops$floating (atom) then
   r := ops$cvaf(atom)
else if ops$integer(atom) then
   r := ops$cvan(atom);

(* Convert a floating-point number to a VAX OPS5 atom *)

square_root := OPS$CVFA (sqrt (r));

end;

end.
```

Use the following statement to link any of the functions on the previous pages to the program STATISTICS.OPS:

```
LINK STATISTICS,FNSQR,OPSINTERP/OPTIONS
```

6.7.2 A VAX OPS5 Program That Calls an External Subroutine

This section contains a VAX OPS5 program that analyzes the daily stock extract and gives you advice. The program calls an external subroutine that adds elements to working memory. The subroutine is written in VAX BASIC.

6.7.2.1 OPS5 Program—STOCK.OPS

```
; STOCK.OPS

(EXTERNAL (READ_NYSE_EXTRACT))      ; Declare the call-out procedure

(LITERALIZE EXTRACTED_LISTING
     STOCK                        ; Name from data file
     CLOSING                      ; Closing price from data file
     HI_LIMIT                     ; Computed price limit
     LO_LIMIT)                    ; Computed price limit

(LITERALIZE REQUEST
     STOCK_NAME                   ; Name requested
     BUY_PRICE)                   ; Original value of stock

(STARTUP
     (MAKE START))
```

```
(P INIT
    { <START> (START) }
  -->
    (WRITE (CRLF) |This program analyzes the daily stock extract |
           (CRLF) |and gives you advice.  Please enter the stock |
           (CRLF) |name: |)
    (BIND <NAME> (ACCEPT))
    (MAKE REQUEST ^STOCK_NAME <NAME>)
    (CALL READ_NYSE_EXTRACT <NAME>)
    (REMOVE <START>))

(P LOOK_AT_EXTRACTED_DATA
    { <REQUEST> (REQUEST ^STOCK_NAME <STOCK_NAME>) }
    { <EXTRACTED_LISTING> (EXTRACTED_LISTING
                                ^STOCK <STOCK_NAME>
                                ^CLOSING <PRICE>) }
  -->
    (WRITE (CRLF) |Found a listing for the stock.  |
                  |What was your buy price? |)
    (BIND <BUY> (ACCEPT))
    (MODIFY <REQUEST> ^BUY_PRICE <BUY>)
    (MODIFY <EXTRACTED_LISTING>
             ^LO_LIMIT (COMPUTE <BUY> - (<BUY> // 10))
             ^HI_LIMIT (COMPUTE <BUY> + (<BUY> // 10))))

(P SELL_FALLING
    { <REQUEST> (REQUEST ^STOCK_NAME <STOCK_NAME>
                         ^BUY_PRICE { <BUY> <> NIL }) }
    { <EXTRACTED_LISTING> (EXTRACTED_LISTING
                                ^STOCK <STOCK_NAME>
                                ^LO_LIMIT <LO_LIMIT>
                                ^CLOSING < <LO_LIMIT>) }
  -->
    (WRITE (CRLF) |The price has dropped more than 10% since you |
                  |bought the stock.|
           (CRLF) |Suggest that you sell.|
           (CRLF) (CRLF))
    (REMOVE <REQUEST>)
    (REMOVE <EXTRACTED_LISTING>)
    (MAKE START))

(P SELL_PROFIT
    { <REQUEST> (REQUEST ^STOCK_NAME <STOCK_NAME>
                         ^BUY_PRICE { <BUY> <> NIL }) }
    { <EXTRACTED_LISTING> (EXTRACTED_LISTING
                                ^STOCK <STOCK_NAME>
                                ^HI_LIMIT <HI_LIMIT>
                                ^CLOSING > <HI_LIMIT>) }
  -->
    (WRITE (CRLF) |The price has gone up.|
           (CRLF) |You may want to sell and make a profit.|
           (CRLF) (CRLF))
    (REMOVE <REQUEST>)
    (REMOVE <EXTRACTED_LISTING>)
    (MAKE START))

(P NOT_ACTIVE
    { <REQUEST> (REQUEST ^STOCK_NAME <STOCK_NAME>
                         ^BUY_PRICE { <BUY> <> NIL }) }
    { <EXTRACTED_LISTING> (EXTRACTED_LISTING
                                ^STOCK <STOCK_NAME> ) }
  -->
    (WRITE (CRLF) |The stock has not had much change in price.|
           (CRLF) |This program can be enhanced to consider other |
                  |factors besides price.|
           (CRLF) (CRLF))
    (REMOVE <REQUEST>)
    (REMOVE <EXTRACTED_LISTING>)
    (MAKE START))
```

```
(P RE_START
    (REQUEST)
  -->
    (MAKE START))
```

6.7.2.2 VAX BASIC External Subroutine—STOCKSUB.BAS

```
        ! STOCKSUB.BAS

100     SUB READ_NYSE_EXTRACT

        ! Definitions of VAX OPS5 support routines
        %INCLUDE "OPSDEF.BAS"

200     MAP (NYSE_COMPOSITE)                           &
                LONG FIFTY_TWO_WEEKS_HIGH,             &
                LONG FIFTY_TWO_WEEKS_LOW,              &
                STRING STOCK = 6,                      &
                LONG DIV,                              &
                LONG YLD_PERCENT,                      &
                LONG PE_RATIO,                         &
                LONG SALES_100S,                       &
                LONG HIGH,                             &
                LONG LOW,                              &
                LONG CLOSING,                          &
                LONG NET_CHG

        ON ERROR GOTO 19000

        OPEN 'NYSE.DAT' FOR INPUT AS FILE #1%,         &
                ORGANIZATION INDEXED,                  &
                MAP NYSE_COMPOSITE,                    &
                PRIMARY KEY STOCK NODUPLICATES

        DECLARE LONG STKNAME, WME_CLASS, ATOM
        DECLARE STRING STOCK_NAME, CLASS, TOTAL_SHARES, &
                       HI_PRICE, LO_PRICE

300     !**********************************************************
        !
        ! Get argument to find stock record
        !
        !**********************************************************

        ! The stock name should be the only argument in result element
        IF (OPS$PARAMETERCOUNT ()) <> 1%
                 THEN Z$ = ("Enter the name of a stock" ) &
                \ CALL OPS$WARNING (Z$ BY REF, LEN (Z$) BY VALUE) &
                \ EXIT SUB

320     ! Get the VAX OPS5 atom
        ! Get the character string that represents the stock name

        STKNAME = OPS$PARAMETER (1% BY VALUE)
        STOCK_NAME = SPACE$(6%)
        CALL OPS$PNAME(STKNAME BY VALUE, STOCK_NAME BY REF, &
                       6% BY VALUE)

        STOCK_NAME = EDIT$ (STOCK_NAME, 34%)
        GET #1%, KEY #0 EQ STOCK_NAME

400     !**********************************************************
        !
        ! Make a new working-memory element
        !
        !**********************************************************

        ! Start by clearing the result element

        CALL OPS$RESET ()

        ! Then create an atom for the class name and place it in the
        ! first field of the result element
        CLASS = "EXTRACTED_LISTING"
        WME_CLASS = OPS$INTERN (CLASS BY REF, LEN (CLASS) BY VALUE)
        CALL OPS$VALUE BY VALUE (WME_CLASS)

        ! Find the field of the STOCK attribute and place the name
        ! atom in the next field

        Z$ = "STOCK"
        CALL OPS$TAB(OPS$INTERN(Z$ BY REF, LEN(Z$) &
                                                BY VALUE)  BY VALUE)
        CALL OPS$VALUE BY VALUE (STKNAME BY VALUE, LEN &
                                                (STOCK_NAME) BY VALUE)

        ! Convert the closing price to an integer atom, set the field
        ! position to the field of the ^CLOSING attribute, and place
        ! the integer atom in that field

        ATOM = OPS$CVNA (CLOSING BY VALUE)
        Z$ = "CLOSING"
        CALL OPS$TAB(OPS$INTERN(Z$ BY REF, LEN(Z$) BY VALUE) BY VALUE)
        CALL OPS$LITVAL BY VALUE (ATOM)

        ! Copy the contents of the result element to working memory,
        ! creating a new working-memory element

        CALL OPS$ASSERT ()

9000    ! Clean up

        CLOSE #1%
        EXIT SUB

19000   Z$ = ("That stock is not in the data base.")
        CALL OPS$WARNING (Z$ BY REF, LEN (Z$) BY VALUE)
        RESUME 9000

        END SUB
```

6.7.3 A VAX OPS5 Program That Uses a VAX DBMS Data Base

This section contains a VAX OPS5 program that calls a VAX BASIC subroutine that calls VAX DBMS. The BLISS-32 file that generates descriptors for calls between the VAX OPS5 program and the VAX BASIC subroutine, the VAX Common Data Dictionary data description language (DDL) file that defines the data base structure to VAX DBMS, and the include file that supplies the data base description generated by VAX DBMS are also provided.

6.7.3.1 VAX OPS5 Program—DBMS.OPS

```
; DBMS.OPS

(LITERAL
     ID = 2
     CUSTOMER-NAME = 3
     LINE-NUMBER = 3
     LINE-QUANTITY = 4
     LINE-NAME = 5
     STATUS = 6)

(EXTERNAL
     (READ_DBMS)
     (WRITE_DBMS)
     (CALL_DBQ))

(STARTUP
     (CALL CALL_DBQ BIND OPSTEST)
     (CALL CALL_DBQ READY EXCLUSIVE UPDATE))
```

```
(P WRITE-TO-DBMS
    { <STATUS> (^STATUS WRITE-TO-DATABASE) }
  -->
    (CALL WRITE_DBMS (SUBSTR <STATUS> 1 INF))
    (MODIFY <STATUS> ^STATUS IN-DATABASE))

(P READ-FROM-DBMS
    { <READ> (READ ^ID <ID> ^STATUS <STATUS>) }
  -->
    (CALL READ_DBMS <ID> <STATUS>)
    (REMOVE <READ>))
```

6.7.3.2 VAX BASIC External Function—DBMS.BAS

```
        ! DBMS.BAS

1       %SBTTL 'READ_DBMS'

        FUNCTION LONG READ_DBMS

        ! This function reads an order structure from the
        ! currently bound and ready data base.  The function
        ! uses the value of the first argument as the order
        ! id, and places the value of the second argument in
        ! the field of the attribute ^STATUS.

        %INCLUDE "DBMS.INC"
        %INCLUDE "OPS.INC"

        EXTERNAL LONG FUNCTION DBQ_TILEND

        ID_ATOM% = OPS$PARAMETER(1%)            ! The first
                                                !  argument
                                                !  (ORDER ID)
        STATUS_ATOM% = OPS$PARAMETER(2%)        ! The second
                                                !  argument
                                                !  (STATUS)
        IF OPS$SYMBOL(ID_ATOM%)                 ! If the atom is a
            THEN                                !  symbol, you must
            ID$ = SPACE$(50%)                   !  define a local
                                                !  dynamic variable
            X% = OPS$BAS_PNAME(ID_ATOM%,ID$)    ! Put the symbol
                                                !  string into ID$
            ELSE                                ! It is a number
            ID$ = NUM1$(OPS$CVAN(ID_ATOM%))     ! Convert it to a
                                                !  string
        END IF
        IF OPS$SYMBOL(STATUS_ATOM%)             ! If the atom is a
            THEN                                !  symbol, you must
            STATUS$ = SPACE$(50%)               !  define a local
                                                !  dynamic variable
                                                ! Put the symbol
                                                !  string into
                                                !  STATUS
            X% = OPS$BAS_PNAME(STATUS_ATOM%,STATUS$
            ELSE                                ! It is a number
                                                ! Convert it to a
                                                !  string
            STATUS$ = NUM1$(OPS$CVAN(STATUS_ATOM%))
        END IF
        LSET ORDER_ID = ID$                     ! Move ID string to
                                                !  DBMS work area
        STATUS% = DBQ_TILEND("FETCH FIRST ORDER_REC USING ORDER_ID")
                                                ! Try to read record
            IF STATUS%                          ! If record exists,
            THEN
            X% = OPS$RESET()                    ! Clear the result
                                                !  element
```

```
         ! The following code builds a working-memory element like:
         !
         !    (ORDER ^ID <ID> ^STATUS <STATUS> ^CUSTOMER-NAME <NAME>)
         !
         ! Each value call inserts the defined value into the
         ! working-memory element and increments the field.  Each
         ! call to OPS$TAB sets the field for the next value call.
         ! OPS$BAS_INTERN returns (and adds if not present) the
         ! OPS5 symbol value for the defined string.  OPS$LITBIND
         ! returns the field or binding of the attribute defined.

             X% = OPS$VALUE(OPS$BAS_INTERN("ORDER"))
             X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("ID")))
             X% = OPS$VALUE(ID_ATOM%)
             X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("STATUS")))
             X% = OPS$VALUE(STATUS_ATOM%)
             X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("CUSTOMER-NAME")))
             X% = OPS$VALUE(OPS$BAS_INTERN(EDIT$(
                                            ORDER_CUSTOMER_NAME,164%)))
             X% = OPS$ASSERT()                      ! Pass the new
                                                    !  working-memory
                                                    !  element through
                                                    !  the match phase

         ! The following loop makes a working-memory element for each
         ! line-item record encountered.  Note each element starts
         ! with an OPS$RESET and ends with an OPS$ASSERT.

             WHILE DBQ_TILEND("FETCH NEXT LINE_ITEM_REC WITHIN" + &
                                                 "ORDER_LINE_ITEM_SET")
                 X% = OPS$RESET()
                 X% = OPS$VALUE(OPS$BAS_INTERN("LINE-ITEM"))
                 X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("ID")))
                 X% = OPS$VALUE(ID_ATOM%)
                 X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("STATUS")))
                 X% = OPS$VALUE(STATUS_ATOM%)
                 X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN("LINE-NAME")))
                 X% = OPS$VALUE(OPS$BAS_INTERN(EDIT$(
                                                LINE_ITEM_NAME,164%)))
                 X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN(
                                                "LINE-NUMBER")))
                 X% = OPS$VALUE(OPS$CVNA(LINE_ITEM_NUMBER))
                 X% = OPS$TAB(OPS$LITBIND(OPS$BAS_INTERN(
                                                "LINE-QUANTITY")))
                 X% = OPS$VALUE(OPS$CVNA(LINE_ITEM_QUANTITY))
                 X% = OPS$ASSERT()
             NEXT
         END IF

         END FUNCTION

1000     %SBTTL 'WRITE_DBMS'

         FUNCTION LONG WRITE_DBMS
         ! This routine writes one data base record each time
         ! the routine is called.  The routine is called with
         ! a complete working-memory element.  The routine
         ! looks at the element's class name to decide what
         ! record to write.  Each field in each record is
         ! mapped to an attribute from the working-memory
         ! element and each field value is extracted from the
         ! appropriate field in the element.  This routine
         ! expects the data base to be bound and ready and the
         ! calls to be made in the correct sequence (that is,
         ! order before its line items).

         %INCLUDE "DBMS.INC"
         %INCLUDE "OPS.INC"

         EXTERNAL LONG FUNCTION DBQ_TILEND
```

```
WME_TYPE$ = SPACE$(50%)
                                            ! Get the class name
X% = OPS$BAS_PNAME(OPS$PARAMETER(1%),WME_TYPE$)
SELECT WME_TYPE$                            ! Select by class
                                            !  name
    CASE = "ORDER"                          ! If it is an order,
                                            !  get the value for
                                            !  attribute ^ID
        X% = OPS$PARAMETER(OPS$CVAN(OPS$LITBIND(
                                   OPS$BAS_INTERN("ID"))))
        IF OPS$SYMBOL(X%)
            THEN
                X$ = SPACE$(50%)
                X% = OPS$BAS_PNAME(X%,X$)
                LSET ORDER_ID = X$          ! Put the value in
                                            !  ORDER_ID
            ELSE
                LSET ORDER_ID = NUM1$(OPS$CVAN(X%))
        END IF
                                            ! Attribute
                                            !  CUSTOMER-NAME &
        X% = OPS$PARAMETER(OPS$CVAN(OPS$LITBIND(
                        OPS$BAS_INTERN("CUSTOMER-NAME"))))
        IF OPS$SYMBOL(X%)
            THEN
                X$ = SPACE$(50%)
                X% = OPS$BAS_PNAME(X%,X$)
                                            !  into
                                            !  ORDER_CUSTOMER_
                                            !  NAME
                LSET ORDER_CUSTOMER_NAME = X$
            ELSE
                LSET ORDER_CUSTOMER_NAME = NUM1$(OPS$CVAN(X%))
        END IF
                                            ! Write it
        STATUS% = DBQ_TILEND("STORE ORDER_REC")
    CASE = "LINE-ITEM"                      ! LINE-ITEM rec
                                            ! Attribute
                                            !  ^LINE-NAME &
        X% = OPS$PARAMETER(OPS$CVAN(OPS$LITBIND(
                          OPS$BAS_INTERN("LINE-NAME"))))
        IF OPS$SYMBOL(X%)
            THEN
                X$ = SPACE$(50%)
                X% = OPS$BAS_PNAME(X%,X$)
                                            !  into
                LSET LINE_ITEM_NAME = X$    !  LINE_ITEM_NAME
            ELSE
                LSET LINE_ITEM_NAME = NUM1$(OPS$CVAN(X%))
        END IF
                                            ! Attribute
                                            !  ^LINE-NUMBER &
        X% = OPS$PARAMETER(OPS$CVAN(OPS$LITBIND(
                         OPS$BAS_INTERN("LINE-NUMBER"))))
        IF OPS$SYMBOL(X%)
            THEN
                LINE_ITEM_NUMBER = 0
            ELSE
                                            !  into
                                            !  LINE_ITEM_NUMBER
                LINE_ITEM_NUMBER = OPS$CVAN(X%)
        END IF
                                            ! Attribute
                                            !  ^LINE-QUANTITY &
        X% = OPS$PARAMETER(OPS$CVAN(OPS$LITBIND(
                       OPS$BAS_INTERN("LINE-QUANTITY"))))
        IF OPS$SYMBOL(X%)
            THEN
```

```
                                        !  into
                LINE_ITEM_QUANTITY = 0  !  LINE_ITEM_
                                        !  QUANTITY
        ELSE
                LINE_ITEM_QUANTITY = OPS$CVAN(X%)
        END IF
        STATUS% = DBQ_TILEND("STORE LINE_ITEM_REC")

END SELECT

END FUNCTION

2000    %SBTTL 'CALL_DBQ'

        FUNCTION LONG CALL_DBQ

        %INCLUDE "DBMS.INC"
        %INCLUDE "OPS.INC"

        EXTERNAL LONG FUNCTION DBQ_TILEND

        PARAM_COUNT% = OPS$PARAMETERCOUNT()
        CALL_STRING$ = ""
        FOR X% = 1% TO PARAM_COUNT%
            ATOM% = OPS$PARAMETER(X%)
            IF OPS$SYMBOL(ATOM%)
                THEN
                X$ = SPACE$(50%)
                Y% = OPS$BAS_PNAME(ATOM%,X$)
                CALL_STRING$ = CALL_STRING$ + X$ + " "
                ELSE
                CALL_STRING$ = CALL_STRING$ +  &
                                NUM1$(OPS$CVAN(ATOM%)) + " "
            END IF
        NEXT X%
        STATUS% = DBQ_TILEND(CALL_STRING$)

        END FUNCTION

20000   %SBTTL 'DBQ_TILEND'

        FUNCTION LONG DBQ_TILEND (STRING DBQS)

        ! This function controls iterative reads supplied by
        ! a data base.  The function returns true while
        ! successful read operations are done and returns
        ! false when the end condition is returned from DBMS.
        ! All other DBMS errors are treated as fatal.

        EXTERNAL LONG FUNCTION &
                        DBQ$INTERPRET,          ! VAX/VMS DBMS
                                                !  function &
                        LIB$MATCH_COND          ! VAX/VMS RTL
                                                !  function

        EXTERNAL LONG CONSTANT SS$_NORMAL, DBM$_END

        DECLARE LONG DBQ_MATCH, DBQ_RETSTS
        DBQ_MATCH = 0%
        DBQ_RETSTS = DBQ$INTERPRET(DBQS)
        IF DBQ_RETSTS AND SS$_NORMAL THEN       ! Check for OK first
                DBQ_TILEND = -1%                ! Hopefully OK will
                EXIT FUNCTION                   !  be the case the
                                                !  majority of the
                                                !  time
        ELSE
                DBQ_MATCH = LIB$MATCH_COND(DBQ_RETSTS, DBM$_END)
                IF DBQ_MATCH = 1% THEN
                        DBQ_TILEND = 0%
                        EXIT FUNCTION
                ELSE
                        CALL LIB$SIGNAL(DBQ_RETSTS BY VALUE)
                END IF
        END IF
        END FUNCTION
```

6.7.3.3 VAX OPS5 Include File—OPS.INC

```
! OPS.INC

! This file contains declarations for the most common VAX OPS5
! support routines.  All VAX OPS5 support routines expect
! arguments to be passed by value, which is why string
! arguments require an intermediate function to be called.

EXTERNAL LONG FUNCTION &
                OPS$BAS_PNAME,                  &
                OPS$BAS_INTERN,                 &
                OPS$ACCEPT,                     &
                OPS$ACCEPTLINE,                 &
                OPS$ASSERT,                     &
                OPS$ATOM,                       &
                OPS$CVAN(LONG BY VALUE),        &
                OPS$CVNA(LONG BY VALUE),        &
                OPS$LITBIND(LONG BY VALUE),     &
                OPS$PARAMETER(LONG BY VALUE),   &
                OPS$PARAMETERCOUNT,             &
                OPS$RESET,                      &
                OPS$SYMBOL(LONG BY VALUE),      &
                OPS$TAB(LONG BY VALUE),         &
                OPS$VALUE(LONG BY VALUE)
```

6.7.3.4 BLISS-32 File—OPSATOM.B32

```
! OPSATOM.B32

MODULE BASIC_TO_OPS (
                ADDRESSING_MODE(EXTERNAL=GENERAL,
                                NONEXTERNAL=GENERAL)
                ) =
BEGIN

!+
!
!   This module provides the interface between VAX OPS5 and VAX BASIC.
!   The routines build the string descriptor required by VAX BASIC
!   from the appropriate VAX OPS5 values.
!
!-

FIELD
    BASIC_DESCRIPTOR_FIELDS =
        SET
        DESC_LEN = [0,0,16,0],
        DESC_TYPE = [0,16,8,0],
        DESC_CODE = [0,24,8,0],
        DESC_PTR = [1,0,32,0]
        TES;

EXTERNAL ROUTINE
        OPS$INTERN,
        OPS$PNAME;

FORWARD ROUTINE
        OPS$BAS_INTERN,
        OPS$BAS_PNAME;

MACRO
    $BASIC_DESCRIPTOR = BLOCK[2] FIELD (BASIC_DESCRIPTOR_FIELDS) %;

GLOBAL ROUTINE OPS$BAS_INTERN (str) =
BEGIN

MAP str: REF $BASIC_DESCRIPTOR;
  RETURN OPS$INTERN(CH$PTR(.str[DESC_PTR]), .str[DESC_LEN]);
```

```
END;

GLOBAL ROUTINE OPS$BAS_PNAME (atom, str) =
BEGIN

MAP str: REF $BASIC_DESCRIPTOR;
  str[DESC_LEN] = OPS$PNAME(..atom, CH$PTR(.str[DESC_PTR]),
  .str[DESC_LEN])

END;
END
ELUDOM
```

6.7.3.5 VAX Common Data Dictionary Data Description Language File—DBMS.DDL

```
SCHEMA   OPSTEST

AREA     ORDER_FILE

RECORD   ORDER_REC
 WITHIN  ORDER_FILE
         ITEM ORDER_ID                 TYPE CHARACTER 20
         ITEM ORDER_CUSTOMER_NAME      TYPE CHARACTER 20

RECORD   LINE_ITEM_REC
 WITHIN  ORDER_FILE
         ITEM LINE_ITEM_NUMBER         TYPE SIGNED LONGWORD
         ITEM LINE_ITEM_QUANTITY       TYPE SIGNED LONGWORD
         ITEM LINE_ITEM_NAME           TYPE CHARACTER 11

SET      ALL_ORDER_SET
         OWNER           SYSTEM
         MEMBER          ORDER_REC
         INSERTION       AUTOMATIC
         RETENTION       FIXED

SET      ORDER_LINE_ITEM_SET
         OWNER           ORDER_REC
         MEMBER          LINE_ITEM_REC
         INSERTION       AUTOMATIC
         RETENTION       MANDATORY
         ORDER           LAST
```

6.7.3.6 Include File—DBMS.INC

```
! DBMS.INC

MAP (DBM$UWA_B) &
      STRING DBMUWA = 0, &
      WORD DBM_CRID, &
      WORD FILL, &
      STRING DBM_CRNS = 1, &
      STRING DBM_CRNM = 31, &
      LONG DBM_MSGVEC, &
      LONG DBM_COND, &
      STRING FILL = 64, &
      LONG DBM_RTB (3), &
  ! ..... ORDER_REC Record ..... &
      STRING ORDER_ID = 20, &
      STRING ORDER_CUSTOMER_NAME = 20, &
  ! ..... LINE_ITEM_REC Record ..... &
      LONG LINE_ITEM_NUMBER, &
      LONG LINE_ITEM_QUANTITY, &
      STRING LINE_ITEM_NAME = 11, &
      STRING FILL = 1, &
      STRING FILL = 0
```

```
EXTERNAL LONG FUNCTION DBQ$INTERPRET
EXTERNAL LONG FUNCTION DBM$ACCEPT
EXTERNAL LONG FUNCTION DBM$PLACE
EXTERNAL LONG FUNCTION DBM$SIGNAL
EXTERNAL LONG FUNCTION DBM$STATS
EXTERNAL LONG CONSTANT &
 DBM$_END, &
 SS$_NORMAL
  ! This list of constants is shortened for clarity
```

6.8 Handling an Asynchronous System Trap (AST)

On VMS systems, programs can get information from external sources, such as timers and I/O devices, by calling system routines which let the program request that it be interrupted when a particular event occurs. Since the interrupt occurs asynchronously (out of sequence) with respect to the program's execution, the interrupt mechanism is called an asynchronous system trap (AST). The trap provides a transfer of control to a user-specified procedure that handles the event.

When a program calls a system routine, it specifies the AST handler as one of its arguments. The program then continues to execute while the system routine performs its task independently. When the system routine finishes, it interrupts the calling program by passing control to the AST handler specified. When the AST handler finishes, the program continues from the point where it was interrupted.

Normally, the AST routine examines the result of calling the system routine, possibly updates the program's data, and then returns control to the program at the point the interruption occurred.

In a VAX OPS5 program, however, the data (working memory) can only be updated at certain points in the recognize-act cycle, so ASTs have to be "synchronized." This is done using the OPS$COMPLETION support routine. The AST handler does not alter working memory itself—instead it calls OPS$COMPLETION, passing the address of a routine, called the completion routine, which updates working memory. When the run-time system gets to the point in the recognize-act cycle where it is safe to update working memory, it calls the completion routine.

6.8.1 Synchronizing Completion Routines

For a program to communicate with such external sources, you must create:

- An external routine that accesses the source by calling a VMS system service that will complete asynchronously
- An AST service routine called by the VMS operating system when the information from the external source becomes available. This routine will execute asynchronously, calling the OPS$COMPLETION support routine with the address of a completion routine as an argument.
- A completion routine called by the VAX OPS5 program at a suitable time, which passes information from the external source to the program by creating working-memory elements

NOTE

The completion routine must call the OPS$COMPLETION support routine with the argument value 0 to avoid being called again unnecessarily after the next recognize-act cycle.

Part III
VAX OPS5 Operator, Declaration, Statement, Action, Function, Command, and Support-Routine Descriptions

Part III of this manual describes the VAX OPS5 operators, declarations, statements, actions, functions, commands, and support routines. The descriptions in each chapter are presented alphabetically by name, providing a quick reference tool. The descriptions include usage details, syntax, format, arguments, and examples.

Many of the constructs described in the following chapters can be specified with arguments. When you specify argument values with an operator, declaration, statement, action, function, or command, separate the values with any combination of spaces, tabs, and carriage returns. When you specify arguments with support routines, separate the values with a comma and a space.

Chapter 7

Operators

This chapter describes the VAX OPS5 operators. Operators are characters that belong to the ASCII character set and have specific meaning to the VAX OPS5 syntax. Table 7–1 lists the operators and provides brief descriptions. The rest of the chapter provides detailed descriptions and examples.

When you specify argument values with an operator, separate them with any combination of spaces, tabs, and carriage returns, unless specified otherwise.

Table 7–1: Summary of Operators

Operator	Description
^	Specifies an attribute of a working-memory element
=	Produces a match if an atom in a working-memory element is the same type as and equal to a specified value
<>	Produces a match if an atom in a working-memory element is not the same type as or not equal to a specified value
>	Produces a match if an atom in a working-memory element is the same type as and greater than a specified value
>=	Produces a match if an atom in a working-memory element is the same type as and greater than or equal to a specified value
<	Produces a match if an atom in a working-memory element is the same type as and less than a specified value
<=	Produces a match if an atom in a working-memory element is the same type as and less than or equal to a specified value
<=>	Produces a match if an atom in a working-memory element is the same type as a specified value
{ }	Specifies a conjunction (logical AND)
<< >>	Specifies a disjunction (logical OR)
//	Prevents evaluation of symbols, operators, variables, and function calls
\ \	Forces evaluation of symbols, operators, variables, and function calls

^

Specifies an attribute of a working-memory element. You can specify attributes in condition elements and actions.

For more information about attributes, see Section 2.1.2.

Format

^attribute-name

Argument

attribute-name
A symbolic attribute name must be declared in a LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration before it is used.

An integer indicates the field to which the run-time system is to refer. Suppose you want an attribute to refer to the atom stored in the second field of a working-memory element. You can specify:

```
^2
```

In actions, you can represent an attribute name with a variable that is bound to a declared attribute name or an integer. If a variable is bound to an attribute name, the run-time system refers to the field assigned to that name. Likewise, if the variable is bound to an integer, the system refers to that field.

Example

Suppose a LITERALIZE declaration assigns fields 2, 3, and 4 to the attribute names NUMBER, AMOUNT, and COUNTED, respectively. Consider the following condition element:

```
(CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO)
```

A working-memory element that stores the atoms 102, 10.06, and NO as follows will match the condition element:

- 102 in field 2
- 10.06 in field 3
- NO in field 4

Suppose a LITERALIZE declaration assigns fields 2, 3, and 4 to the attribute names NUMBER, AMOUNT, and COUNTED, respectively. Suppose also that the variable <NUMBER> is bound to the attribute name NUMBER. Now consider the following MAKE action:

```
(MAKE CHECK ^<NUMBER> 102 ^AMOUNT 10.06 ^COUNTED NO)
```

The action creates a working-memory element whose atoms are stored as follows:

- CHECK (class name) in field 1
- 102 in field 2
- 10.06 in field 3
- NO in field 4

=

Produces a match if an atom in a working-memory element is the same type as and equal to a value specified in a condition element. If you omit the operator from a condition element, the run-time system uses the equal operator to compare the atoms in working-memory elements with that value.

The equal operator is the only predicate that can precede the first occurrence of a variable, because the first time you use a variable, the run-time system binds the variable to a value.

For more information about predicates, see Section 3.2.1.3.

Format

= *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a constant, a variable, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is the integer 102 will match the following condition element:

```
(CHECK ^NUMBER = 102)
```

As the run-time system uses the equal operator implicitly if you do not specify a predicate, the following condition element is equivalent to the preceding element:

```
(CHECK ^NUMBER 102)
```

<>

Produces a match if an atom in a working-memory element is not the same type as or not equal to a value specified in a condition element.

Format

<> *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a constant, a variable bound to a constant, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is not the integer 102 will match the following condition element:

```
(CHECK ^NUMBER <> 102)
```

>

Produces a match if an atom in a working-memory element is the same type as and greater than a value specified in a condition element.

Format

> *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a number, a variable bound to a number, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer greater than 102 will match the following condition element:

```
(CHECK ^NUMBER > 102)
```

>=

Produces a match if an atom in a working-memory element is the same type as and greater than or equal to a value specified in a condition element.

Format

>= *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a number, a variable bound to a number, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer greater than or equal to 102 will match the following condition element:

```
(CHECK ^NUMBER >= 102)
```

<

Produces a match if an atom in a working-memory element is the same type as and less than a value specified in a condition element.

Format

< *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a number, a variable bound to a number, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer less than 102 will match the following condition element:

```
(CHECK ^NUMBER < 102)
```

<=

Produces a match if an atom in a working-memory element is the same type as and less than or equal to a value specified in a condition element.

Format

<= *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a number, a variable bound to a number, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer less than or equal to 102 will match the following condition element:

```
(CHECK ^NUMBER <= 102)
```

<=>

Produces a match if an atom in a working-memory element is the same type as a value specified in a condition element. The types of values are symbols, integers, and floating-point numbers. For example, if you specify the operator with a symbol or a variable bound to a symbol, a match is produced if the atom in the working-memory element is a symbol.

Format

<=> *value*

Argument

value
The value to which an atom in a working-memory element is to be compared. The value must be a number, a variable bound to a number, or a function call.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer will match the following condition element:

```
(CHECK ^NUMBER <=> 102)
```

{ }

Specifies a conjunction. A conjunction is similar to a logical AND. It is a left-hand-side pattern containing one or more conditional tests, all of which an atom in a working-memory element must satisfy.

For more information about conjunctions, see Section 3.2.1.4.

Format

{[conditional-test...]}

Argument

conditional-test
One or more conditional tests that an atom in a working-memory element is to satisfy.

The argument is optional. If you do not specify a conditional test, the braces act as a condition-element placeholder, that is, they indicate the presence of an atom whose value you do not want to test. Use a placeholder to skip over atoms in a vector attribute's value, or when you specify a condition element without using attribute names.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is an integer between 102 and 105 will match the following condition element:

```
(CHECK { ^NUMBER >= 102 <= 105 })
```

<< >>

Specifies a disjunction. A disjunction is similar to a logical inclusive OR. It is a left-hand-side pattern containing a list of constant values, one of which can match the pattern.

For more information about disjunctions, see Section 3.2.1.5.

Format

<< *value...* **>>**

Argument

value
A symbol, integer, or floating-point number that an atom in a working-memory element is to match. You can specify one or more values.

The atoms you specify in a disjunction are not evaluated; therefore, operators and variables are recognized as symbols.

Example

A working-memory element that has the class name CHECK and the attribute ^NUMBER whose value is the integer 103, 105, or 108 will match the following condition element:

```
(CHECK ^NUMBER << 103 105 108 >>)
```

//

Prevents the evaluation of symbols, operators, variables, and function calls. You can use this operator (quotes) in condition elements and actions.

Format

// value

Argument

value
The symbol, operator, variable, or function call to be quoted.

Example

To find an atom in a working-memory element that matches the symbol <NUMBER>, specify the following condition element:

```
(CHECK ^NUMBER // <NUMBER>)
```

The atom <NUMBER> in a working-memory element will match the symbol <NUMBER>. If you do not use the quote operator (//), the atom will match the value bound to the variable <NUMBER>.

\\

Forces the evaluation (unquotes) of symbols, operators, variables, and function calls. Use this operator to unquote values in the condition elements and actions you specify as arguments in a BUILD action. You must remove the quote for each value you want the BUILD action to evaluate. For more information about the BUILD action, see the description in Chapter 10.

Format

\\ *value*

Argument

value
The symbol, operator, variable, or function call to be evaluated.

Example

Suppose the variable <CITY> in the following BUILD action is bound to BOSTON:

```
(BUILD ADD-ADDRESS
     (HOUSE ^CITY \\<CITY> ^ADDRESS <ADDRESS>)
   -->
     (WRITE |New listing -- | <ADDRESS>))
```

The new production is:

```
(P ADD-ADDRESS
     (HOUSE ^CITY BOSTON ^ADDRESS <ADDRESS>)
   -->
     (WRITE |New listing -- | <ADDRESS>))
```

The unquoted variable <CITY> is evaluated to BOSTON. However, the variable <ADDRESS> is not evaluated until the production ADD–ADDRESS is executed during another recognize-act cycle.

Chapter 8
Declarations

This chapter describes the VAX OPS5 declarations. Table 8–1 lists the names of the declarations and gives brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name. For more information about declarations, see Section 2.3.

When you specify argument values in a declaration, separate them with any combination of spaces, tabs, and carriage returns.

Table 8–1: Summary of Declarations

Declaration	Description
EXTERNAL	Declares external routines
LITERAL	Assigns specified fields of a working-memory element to attribute names
LITERALIZE	Associates a class with a list of attribute names and assigns fields of a working-memory element to the specified attribute names
VECTOR–ATTRIBUTE	Assigns a predefined field of a working-memory element to specified vector-attribute names

EXTERNAL

Declares external routines, which are functions or subroutines written in VAX languages other than VAX OPS5. The EXTERNAL declaration provides the names of the routines to the VAX OPS5 compiler.

There are two sorts of EXTERNAL declarations (Format A and Format B). The mechanism you must use for returning values from external functions depends on which format is used.

For more information about declaring and calling external routines, see Chapter 6 in this document, the *VAX Architecture Handbook*, and the *Introduction to VMS System Routines*.

Format A

EXTERNAL *{(external-routine-spec)}...*

where external-routine-spec is:

external-routine-name [return-type] [([argument-type] [mechanism])]...

Arguments

external-routine-name
A symbol that represents the name of an external routine. The name you specify must be the same as the name specified in the external routine's code. You can specify one or more routine names.

return-type
If this is present, it must be one of the symbols SYMBOLIC–ATOM, NUMERIC–ATOM, FLOAT–ATOM, INTEGER–ATOM, or ANY–ATOM. It indicates which type of atom is expected to be returned by the external routine that you are declaring, and may cause a run-time check to be performed.

argument-type
If this is present, it must be one of the symbols SYMBOLIC–ATOM, NUMERIC–ATOM, FLOAT–ATOM, INTEGER–ATOM, or ANY–ATOM. It indicates the type of value that should be used for the corresponding routine argument, and may cause a run-time check to be performed. If you do not specify the argument-type, ANY–ATOM is used as the default.

mechanism
This argument indicates the passing mechanism to be used. If it is present, it must be either BY VALUE (which causes the value of the atom to be passed), or BY REFERENCE (which causes the atom's address to be passed). If you do not specify the mechanism, BY VALUE is used as the default.

NOTES

The mechanism argument and the return-type argument are ignored if the routine is used in a CALL action.

If the external routine is called as a function, it must use the standard VMS mechanism to return a value, that is, the value must be in register R0.

Example

The following EXTERNAL declaration declares the external routines SORT_TRANSACTIONS and BALANCE, and specifies two parameters to be passed for SORT_TRANSACTIONS and one for BALANCE:

```
(EXTERNAL
        (SORT_TRANSACTIONS
           (ANY-ATOM BY VALUE)
           (FLOAT-ATOM BY REFERENCE))
        (BALANCE
           (ANY-ATOM BY REFERENCE)))
```

Format B

EXTERNAL *external-routine-name...*

This syntax has been retained in order to be compatible with earlier versions of VAX OPS5, but you are advised to use format A wherever possible.

Argument

external-routine-name
A symbol that represents the name of an external routine. The name you specify must be the same as the name specified in the external routine's code. You can specify one or more routine names.

NOTES

If the external routine is called as a function on the left-hand side of a production, it must use the standard VMS mechanism to return a value, that is, the value must be in register R0.

If the external routine is called as a function on the right-hand side of a production, it can only return a value by means of a call to the OPS$VALUE routine described in Chapter 13.

Calls to OPS$VALUE will have no effect if they are made from an external function that was called from the left-hand side of a production.

Example

The following EXTERNAL declaration declares the external routines SORT_ TRANSACTIONS and BALANCE:

```
(EXTERNAL
     SORT_TRANSACTIONS
     BALANCE)
```

LITERAL

Assigns specified fields of a working-memory element to attribute names. The LITERAL declaration lets you control which field the compiler assigns to an attribute.

All declarations must appear before any other type of statement. If the declaration part of a program contains both LITERAL and LITERALIZE declarations, the compiler processes the LITERAL declarations first, regardless of the order in which they are specified. The compiler uses the field assignments for the attributes specified in both LITERAL and LITERALIZE declarations. If the assignments specified by a LITERAL declaration conflict with assignments made by a LITERALIZE declaration, the compiler displays the following message:

```
%OPSCOMP-W-LITCLASH, Literal value clash involving AAAAAA in
literalize declaration -- old value kept
```

Format

LITERAL *{attribute-name = field}...*

Arguments

attribute-name
The name of an attribute to which a field is to be assigned.

field
An integer that indicates the field assigned to the specified attribute.

Example

The following LITERAL declaration assigns the fields 2, 3, and 4 to the attribute names NUMBER, AMOUNT, and COUNTED, respectively:

```
(LITERAL
     NUMBER = 2
     AMOUNT = 3
     COUNTED = 4)
```

LITERALIZE

Associates a class with a list of attribute names and assigns fields of a working-memory element to the specified attribute names.

All declarations must appear before any other type of statement. If the declaration part of a program contains both LITERAL and LITERALIZE declarations, the compiler processes the LITERAL declarations first, regardless of the order in which they are specified. The compiler uses the field assignments for the attributes specified in both LITERAL and LITERALIZE declarations. If the assignments specified by a LITERAL declaration conflict with assignments made by a LITERALIZE declaration, the compiler displays the following message:

```
%OPSCOMP-W-LITCLASH, Literal value clash involving AAAAAA in
literalize declaration -- old value kept
```

Format

LITERALIZE *class-name attribute-name...*

Arguments

class-name
The name of the class with which the specified attribute names are to be associated.

attribute-name
The name of an attribute to which a field is to be assigned. You can specify one or more attribute names, of which only one can be the name of a vector attribute.

Example

The following LITERALIZE declaration associates the attribute names NUMBER, AMOUNT, COUNTED, and DATE with the class name CHECK and assigns fields to the attribute names:

```
(LITERALIZE CHECK
     NUMBER
     AMOUNT
     COUNTED
     DATE)
```

VECTOR–ATTRIBUTE

Assigns field 256 of a working-memory element to specified vector-attribute names. The first atom of a vector attribute's value is stored in that field.

You can declare all vector attributes for a program either separately or in one declaration. All vector-attribute declarations must be given before the STARTUP statement (if any) and before the first production in the program.

NOTE

Vector-attribute declarations are global within a program. Once you have declared the name of a vector attribute, you cannot use the same name for a scalar attribute.

Format

VECTOR–ATTRIBUTE *attribute-name...*

Argument

attribute-name
The name of a vector attribute to be assigned the predefined field for vector attributes. You can specify the name of one or more vector attributes.

Example

The following VECTOR–ATTRIBUTE declaration declares vector attributes named DATE and TRANSACTIONS:

```
(VECTOR-ATTRIBUTE DATE TRANSACTIONS)
```

Chapter 9

Statements

This chapter describes the VAX OPS5 statements. Table 9–1 lists the names of the statements and gives brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name.

When you specify argument values in a statement, separate them with any combination of spaces, tabs, and carriage returns.

Table 9–1: Summary of Statements

Statement	Description
CATCH	Creates a catcher
PRODUCTION	Performs right-hand-side actions when left-hand-side conditions are met
STARTUP	Executes actions and commands that set up initial conditions for a program's execution

CATCH

Creates a catcher, which is a list of actions that are executed after a specified number of recognize-act cycles have been executed. An AFTER action or command specifies the number of recognize-act cycles to be executed before the catcher is executed.

For more information about catchers and controlling loops, see Section 5.9.

Format

CATCH *catcher-name action...*

Arguments

catcher-name
A unique symbol that names the catcher being created. The symbol cannot represent the name of another catcher, a production, an external routine, an action, or a function that already exists in the program.

action
An action. You can specify one or more actions. For information about actions, see Section 3.3 and Chapter 10.

Example

The following CATCH statement creates a catcher named BALANCE, which displays a message after the number of recognize-act cycles specified in an AFTER action or command have been executed, and stops execution:

```
(CATCH BALANCE
     (WRITE (CRLF) |Check your balance.|)
     (HALT))
```

PRODUCTION

Performs right-hand-side actions when left-hand-side conditions are met and the production has been selected from the conflict set.

Format

P *production-name condition-element... action...*

Arguments

production-name
A unique symbol that names the production being created. The symbol cannot represent the name of another production, a catcher, an external routine, an action, or a function that already exists in the program.

condition-element
A specified pattern against which working memory elements can be matched. See Chapter 3 for information about condition elements, and Chapter 4 for information about matching.

Condition elements can have condition-element variables associated with them, can be negative, and can introduce variables that are used in other condition elements and in actions on the right-hand side.

action
An action. You can specify one or more actions. For information about actions, see Section 3.3 and Chapter 10.

Example

The following production, named STOP–COUNT, halts the program and removes the working-memory element bound to the variable <REPLY>:

```
(P STOP-COUNT
     { <REPLY>
       (REPLY ^DATE STOP) }
   -->
     (REMOVE <REPLY>)
     (HALT))
```

STARTUP

Executes actions and commands that are executed before any productions, to set up initial conditions for a program's execution. Only one STARTUP statement can be used in a program, and it is optional.

For more information about initializing VAX OPS5 programs, see Section 5.1.

Format

$$\textbf{STARTUP} \left\{ \begin{array}{l} \textit{action} \\ \textit{command} \end{array} \right\} \ldots$$

Arguments

action
An action. You can specify one or more actions. For information about actions, see Section 3.3 and Chapter 10.

command
A VAX OPS5 command. The commands you can specify are @, DISABLE, ENABLE, RUN, STRATEGY, and WATCH. You can specify one or more commands. For descriptions of the commands, see Chapter 12.

Example

The following STARTUP statement turns off trace output, disables informational messages, sets the conflict-resolution strategy to MEA, creates a working-memory element whose class name is START, and starts the execution of recognize-act cycles:

```
(STARTUP
     (WATCH 0)            ; Disable trace output
     (DISABLE HALT)       ; Disable informational messages
     (STRATEGY MEA)       ; Set conflict-resolution strategy
     (MAKE START)         ; Initialize working memory
     (RUN))               ; Start executing recognize-act cycles
```

Chapter 10

Actions

This chapter provides descriptions of the VAX OPS5 actions. Actions instruct the run-time system to perform operations. Table 10–1 lists the names of the actions and provides brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name. For more information about actions, see Section 3.3.

Most VAX OPS5 actions require at least one argument. You can represent argument values with atoms, variables bound to atoms, or function calls that evaluate to atoms. When you specify argument values, separate them with any combination of spaces, tabs, and carriage returns.

Table 10–1: Summary of Actions

Action	Description
ADDSTATE	Adds the contents of a file produced by the SAVESTATE action or command to the current state of working memory and the conflict set
AFTER	Specifies the number of recognize-act cycles that must be executed before a specified catcher is executed
BIND	Binds a variable to an atom
BUILD	Adds a new production to an executing program
CALL	Calls an external subroutine
CBIND	Binds an element variable to a working-memory element
CLOSEFILE	Closes the open files associated with specified file identifiers and dissociates the identifiers from the files
DEFAULT	Sets the terminal or a file as the default input source for the ACCEPT and ACCEPTLINE functions, or the default output destination for the WRITE action or for trace output
HALT	Stops the run-time system from executing recognize-act cycles at a particular point during a program's execution
MAKE	Creates a working-memory element
MODIFY	Changes one or more atoms in an existing working-memory element
OPENFILE	Opens a file and associates it with a file identifier
REMOVE	Deletes elements from working memory
RESTORESTATE	Clears and then restores working memory and the conflict set to the state recorded in a file produced by the SAVESTATE action or command

(continued on next page)

Table 10–1 (Cont.): Summary of Actions

Action	Description
SAVESTATE	Copies the state of working memory and the conflict set to a file
WRITE	Sends output from a program to the terminal or a file

ADDSTATE

Adds the contents of a file produced by the SAVESTATE action or command to the current state of working memory and the conflict set. The time tags of the working-memory elements in the saved file are ignored.

Format

ADDSTATE *file-spec*

Argument

file-spec
A VMS file specification for a file previously produced by the SAVESTATE action or command.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you specify includes a semicolon, enclose the specification in vertical bars (| |).

Example

Suppose you use the SAVESTATE action to store a state of working memory and the conflict set in the file CHECKS.DAT. The following action adds the contents of the file CHECKS.DAT to the current state of working memory and the conflict set:

```
(ADDSTATE CHECKS.DAT)
```

AFTER

Specifies the number of recognize-act cycles that must be executed before a specified catcher is executed, thus controlling loops in a program.

For more information about catchers and controlling loops, see Section 5.9.

Format

AFTER *cycles catcher-name*

Arguments

cycles
A positive integer that specifies the number of recognize-act cycles that are to be executed before the specified catcher is executed. If execution halts before the specified number of cycles has been executed, the catcher is not executed.

catcher-name
A symbol that names a catcher.

Example

The following action specifies that the catcher named BALANCE is to be executed after five recognize-act cycles have been executed:

```
(AFTER 5 BALANCE)
```

If program execution halts before five cycles have been executed, the catcher is not executed.

BIND

Binds a variable to an atom.

Format

BIND *variable [rhs-expression]*

Arguments

variable
The variable to which an atom is to be bound.

rhs-expression
A right-hand-side expression to be evaluated. The action binds the specified variable to the atom that results from the evaluation.

This argument is optional. If you do not specify the argument, the action uses the GENATOM function to generate an atom, and binds the specified variable to that atom.

Example

Consider the following right-hand-side expression:

```
(COMPUTE <BALANCE> - <WITHDRAWAL>)
```

When evaluated, this expression computes the difference between the values bound to the variables <BALANCE> and <WITHDRAWAL>. The following BIND action evaluates this expression and binds the variable <NEW–BALANCE> to the atom that results:

```
(BIND <NEW-BALANCE> (COMPUTE <BALANCE> - <WITHDRAWAL>))
```

The following action creates a new atom and binds the variable <NEW–ATOM> to that atom:

```
(BIND <NEW-ATOM>)
```

BUILD

Adds a new production to an executing program.

By default, the BUILD action treats variables, actions, and function calls as constants. If you want the action to evaluate a variable, action, or function call when the run-time system adds the production to the program, precede the variable, action, or function call with the unquote operator (\\). This operator is described in Chapter 7.

The source code for the new production is stored in a file named OPS$BUILD.OPS. Each time a BUILD action is executed, the run-time system creates a new version of the file. You can add the productions stored in the build files (OPS$BUILD.OPS;n) to a program's source code by specifying the files when you compile the program, or by editing the program to include the build files.

For more information about adding productions to an executing program, see Section 5.11.

Format

BUILD *production-name left-hand-side --> right-hand-side*

Arguments

production-name
A symbol that names the production to be added to the program. The symbol cannot represent the name of a catcher or an external routine that already exists in the program. If the BUILD action finds an existing production with the name that you have specified, the original production is disabled and the new one is built. However, disabled productions remain in memory; therefore, if you build the same production many times, you decrease system performance.

Do not precede the production name with an open parenthesis and the symbol P. The BUILD action adds these characters when it creates the production.

left-hand-side
One or more condition elements.

-->
A symbol that separates the left-hand side of the production from the right-hand side.

right-hand-side
One or more actions. Enclose each action within parentheses. However, do not place a closing parenthesis for the new production at the end of the last action. The BUILD action adds the closing parenthesis when it creates the production.

Example

Consider the following production:

```
(P ADD-STOP-COUNT
   (STATUS ^READY-TO-STOP YES)
  -->
   (BUILD STOP-COUNT
       { <REPLY>
         (REPLY ^DATE STOP) }
      -->
       (REMOVE <REPLY>)
       (HALT)))
```

This production adds the following production to the executing program:

```
(P STOP-COUNT
   { <REPLY>
     (REPLY ^DATE STOP) }
  -->
   (REMOVE <REPLY>)
   (HALT))
```

Suppose a program creates the following working-memory element:

```
(BUILD-PRODUCTION NEW-PRODUCTION (DATE ^YEAR 1988)
  --> (MAKE NOV 1 ^DAY-OF-WEEK TUESDAY))
```

Suppose the same program contains the following production:

```
(P ADD-PRODUCTION
     { <BUILD-PRODUCTION>
       (BUILD-PRODUCTION) }
   -->
     (BUILD \\ (SUBSTR <BUILD-PRODUCTION> 2 INF))
     (REMOVE <BUILD-PRODUCTION>))
```

The unquote operator causes the BUILD action to evaluate the call to the SUBSTR function, which provides the BUILD action with the necessary arguments to create a new production while the program is executing.

CALL

Calls an external subroutine, which is a routine written in a VAX language other than VAX OPS5.

For information about calling external subroutines, see the *VAX OPS5 User's Guide*.

Format

CALL *external-routine-name [external-routine-argument...]*

Arguments

external-routine-name
The name of the external subroutine to be called. Specify the name of a subroutine that has been declared with the EXTERNAL declaration. Otherwise, the compiler displays the following warning:

```
%OPSCOMP-W-EXTCALL, Subroutine AAAAAA not declared external
```

external-routine-argument
A number or symbol representing the value of an external routine's argument. This argument is optional and can be specified one or more times.

Example

The following EXTERNAL declaration declares an external subroutine named READ_NYSE_EXTRACT:

```
(EXTERNAL READ_NYSE_EXTRACT)
```

The following action calls the subroutine READ_NYSE_EXTRACT with the variable <NAME> as an argument:

```
(CALL READ_NYSE_EXTRACT <NAME>)
```

NOTES

Type checking may be performed at run time for any argument for which there is a corresponding type specification in the EXTERNAL declaration.

Any passing mechanisms specified in the EXTERNAL declaration of the routine are ignored.

CBIND

Binds an element variable to the last element added to working memory by a MAKE, MODIFY, or CALL action.

For information about specifying element variables, see Section 3.2.2.

Format

CBIND *element-variable*

Argument

element-variable
The element variable to be bound to the last element added to working memory.

Example

The following action binds the element variable <COUNTER> to the last element added to working memory:

```
(CBIND <COUNTER>)
```

CLOSEFILE

Closes the open files associated with specified file identifiers and dissociates the identifiers from the files.

Format

CLOSEFILE *file-id...*

Argument

file-id
The file identifier of an open file to be closed. You can specify one or more file identifiers.

Example

The following action closes the open files associated with the file identifiers CHECKSI and CHECKSO:

```
(CLOSEFILE CHECKSI CHECKSO)
```

DEFAULT

Sets the terminal or a file as the default input source for the ACCEPT and ACCEPTLINE functions, or the default output destination for the WRITE action or trace output. If you do not use the DEFAULT action to specify otherwise, the default source for input and destination for output are SYS$INPUT and SYS$OUTPUT.

Format

DEFAULT *location keyword*

Arguments

location
The source from which input is to be read or the destination to which output is to be written. The value can be either a file identifier or the symbol NIL. If you specify a file identifier, the DEFAULT action sets the source or destination to the open file associated with that name. If you specify NIL, the input is read from or output is sent to the terminal.

keyword
A keyword that specifies whether the default is to be set for the ACCEPT and ACCEPTLINE functions, the WRITE action, or trace output. Table 10–2 lists the keywords you can specify.

Table 10–2: DEFAULT Action Keywords

Keyword	Description
ACCEPT	Input read by the ACCEPT and ACCEPTLINE functions is read from the specified source.
TRACE	Trace output is sent to the specified destination. You can enable and disable trace output, using the WATCH command (see Chapter 12).
WRITE	Output produced by the WRITE action is sent to the specified destination.

Example

The following action sets the open file associated with the file identifier CHECKSI to be the default source of input for the ACCEPT and ACCEPTLINE functions:

```
(DEFAULT CHECKSI ACCEPT)
```

To set the default back to the terminal, specify:

```
(DEFAULT NIL ACCEPT)
```

HALT

Stops the run-time system from executing recognize-act cycles after the current recognize-act cycle ends. If informational messages are enabled, the run-time system displays the following message and invokes the command interpreter:

```
%OPSRT-I-HALTED, HALT -- right-hand-side action

OPS5>
```

If informational messages are disabled, the run-time system returns control to the operating system, or to the calling program if the VAX OPS5 program was called as a subroutine. The ENABLE and DISABLE commands are described in Chapter 12.

Format

HALT

Example

The following production causes the run-time system to stop executing the recognize-act cycles when a working-memory element matches the condition element (REPLY ^DATE STOP):

```
(P STOP-COUNT
   { <REPLY>
     (REPLY ^DATE STOP) }
  -->
   (REMOVE <REPLY>)
   (HALT))

%OPSRT-I-HALTED, HALT -- right-hand-side action

OPS5>
```

MAKE

Creates a working-memory element.

For information about working-memory elements, see Section 2.1.

Format

MAKE *[class-name] [{[scalar-attribute] value}...] [vector-attribute value]*

Arguments

class-name
A symbol that names the class of the element to be created. This argument is optional. If you specify a class name, the action places the name in the first field of the working-memory element. If you do not specify a class name, the action places NIL in the first field of the element.

scalar-attribute
A scalar attribute that describes a characteristic of the working-memory element to be created and specifies the field of the working-memory element in which the corresponding value will be placed. You must specify a value with each scalar attribute.

value
An atom to be placed in the field of the working-memory element indicated by the corresponding attribute.

vector-attribute
A vector attribute that describes a characteristic of the working-memory element to be created. Only one vector attribute can be specified in a MAKE action; if you specify a vector attribute, you must specify a value with it.

value
One or more atoms to be placed in the working-memory element. The first atom is placed in the field indicated by the vector attribute. If the vector attribute is declared, the first atom is placed in field 256, and the remaining atoms are placed sequentially in the fields that follow.

Example

The following production contains two MAKE actions:

```
(P WHAT-DATE
    { <START> (START) }
  -->
    (REMOVE <START>)
    (WRITE (CRLF) (CRLF) |What date do you want to search for?|)
    (WRITE (CRLF) (CRLF) |Enter the day, the first three|
                         |letters of the month, and the year.|
                  (CRLF)
                         |For example -- 14 NOV 1988|
            (CRLF) (CRLF)
                         |Type STOP to halt the program.|
            (CRLF) (CRLF)
                         |Date>>> |)
    (MAKE COUNT ^VALUE 0)
    (MAKE REPLY ^DATE (ACCEPTLINE)))
```

The first MAKE action creates a working-memory element that has the class name COUNT and the attribute ^VALUE whose value is 0.

```
6 [WHAT-DATE] (COUNT ^VALUE 0)
```

The second MAKE action creates a working-memory element that has the class name REPLY and the vector attribute ^DATE whose value is read from the terminal by the ACCEPTLINE function. For example, if the function reads the atoms 14, NOV, and 1988, this MAKE action creates the following element:

```
7 [WHAT-DATE] (REPLY ^DATE 14 NOV 1988)
```

MODIFY

Changes one or more atoms in an existing working-memory element. The action removes an element from working memory and uses the atoms in that element and new atoms that you specify to create a new element. Therefore, when you modify a working-memory element, the element's time tag changes.

You can change more than one atom in a MODIFY action, and you can use the same working-memory element in more than one MODIFY action in the same production.

For more information about working-memory elements, see Section 2.1.

Format

MODIFY *element-designator {attribute value}...*

Arguments

element-designator
An element variable or an integer that refers to a condition element on the left-hand side of the production, which indicates the working-memory element whose atoms are to be changed.

attribute
An attribute that specifies which atom in the working-memory element is to be changed. You must specify a value with each attribute.

value
An atom or a list of atoms (if you specify a vector attribute) to replace one or more atoms in the working-memory element. The corresponding attribute indicates the field in which the new atom is placed.

Example

Suppose working memory contains the following elements:

```
1[NIL](CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2[NIL](CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3[NIL](CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4[NIL](CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6[WHAT-DATE](COUNT ^VALUE 0)
7[WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

The following production contains two MODIFY actions:

```
(P FIND-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
     { <CHECK>
       (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
              ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
```

```
        { <COUNTER>
          (COUNT ^VALUE <VALUE>) }
      -->
        (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                             |for $| <AMOUNT>
                             |dated| (SUBSTR <REPLY> DATE INF))
        (MODIFY <CHECK> ^COUNTED YES)
        (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The first MODIFY action modifies the working-memory element bound to the element variable <CHECK> by changing the value of the attribute ^COUNTED to YES.

The second MODIFY action modifies the working-memory element bound to the element variable <COUNTER> by changing the value of the attribute ^VALUE to the result of the function call (COMPUTE 1 + <VALUE>).

When this production is executed, working memory changes as follows:

```
1[NIL](CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2[NIL](CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
4[NIL](CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
7[WHAT-DATE](REPLY ^DATE 14 NOV 1988)
9[FIND-CHECKS](CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED YES ^DATE 14 NOV 1988)
10[FIND-CHECKS] (COUNT ^VALUE 1)
```

The working-memory element whose time tag is 3 is replaced by the working-memory element whose time tag is 9, and the new value of the attribute ^COUNTED is YES. The element whose time tag is 6 is replaced by the element whose time tag is 10, and the new value of the attribute ^VALUE is 1.

OPENFILE

Opens a file and associates it with a file identifier.

Format

OPENFILE *file-id file-spec keyword*

Arguments

file-id
A symbol that represents the file identifier with which the specified file is to be associated.

file-spec
The VMS file specification for the file to be opened. If you are opening a file for input, the file must already exist.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you specify includes a semicolon, enclose the specification in vertical bars (| |).

keyword
A keyword that indicates whether the specified file is to be opened for input or output. If you specify IN, the action opens an existing file for reading only. If you specify OUT, the action creates a new file and opens it for writing only. If you specify APPEND, the action opens an existing file for writing, and sets the file pointer to the end of the file.

Example

The following action opens the file CHECKS.DAT for input and associates it with the file identifier CHECKSI:

```
(OPENFILE CHECKSI CHECK.DAT IN)
```

REMOVE

Deletes elements from working memory.

NOTE

The binding between a working-memory element and a condition element is not removed until the production's execution is complete, or a RESTORESTATE action or OPS$CLEAR routine is executed.

Format

REMOVE *element-designator...*

Argument

element-designator
An element variable or an integer that refers to a condition element on the left-hand side of the production, which indicates the working-memory element to be deleted. You can specify one or more designators.

Example

Consider the following production:

```
(P COUNTED-CHECKS
     { <REPLY>
       (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
      -(CHECK ^DATE <DAY> <MONTH> <YEAR> ^COUNTED NO)
     { <COUNTER>
       (COUNT ^VALUE <VALUE>) }
   -->
     (REMOVE <REPLY>)
     (REMOVE <COUNTER>)
     (MAKE START)
     (WRITE (CRLF) (CRLF) |There are| <VALUE> |checks dated|
                         <DAY> <MONTH> <YEAR> (CRLF)))
```

The REMOVE actions delete the working-memory elements bound to the element variables <REPLY> and <COUNTER>.

RESTORESTATE

Clears and then restores working memory and the conflict set to the state recorded in a file produced by the SAVESTATE action or command.

Format

RESTORESTATE *file-spec*

Argument

file-spec
The VMS file specification for a file previously produced by the SAVESTATE action or command. The action uses the contents of the file to restore the state of working memory and the conflict set.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you specify includes a semicolon, enclose the specification in vertical bars (| |).

Example

The following action clears and then restores the contents of working memory and the conflict set to the same state recorded in the file CHECKS.DAT:

```
(RESTORESTATE CHECKS.DAT)
```

NOTES

RESTORESTATE clears all working memory elements before loading the new state; therefore, variable bindings are lost.

After a RESTORESTATE action, GENATOM will produce atoms that are different from any that were recorded in the saved file but may repeat atoms that were generated before the RESTORESTATE action was executed.

SAVESTATE

Copies to a file the state of working memory and the conflict set. You can later use the ADDSTATE and RESTORESTATE action or command to add to or overwrite the current memory with the contents of the file.

Format

SAVESTATE *file-spec*

Argument

file-spec
The VMS file specification for the file to which the action is to copy the state of working memory and the conflict set.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you specify includes a semicolon, enclose the specification in vertical bars (| |).

Example

The following action copies the state of working memory and the conflict set to the file CHECKS.DAT:

```
(SAVESTATE CHECKS.DAT)
```

WRITE

Sends output from a program to the terminal or a file. By default, the WRITE action sends output to the terminal. To send output to a file, you can do one of the following:

- Specify the WRITE action with a file identifier
- Change the default destination for the WRITE action, using the DEFAULT action or command

Format

WRITE *[file-id] rhs-expression*

Arguments

file-id
The file identifier of the destination file for the WRITE action's output. This argument is optional. If you do not specify the argument or if the name you specify is not associated with an open output file, the output is sent to the current default for the WRITE action (set with the DEFAULT action or command).

rhs-expression
A right-hand-side expression that represents the output. The action evaluates the expression and sends the output to the terminal or a file. Use the following functions to format the output:

- CRLF —Carriage return/line feed
- TABTO—Tab
- RJUST—Right justify

If you do not use these functions, the WRITE action displays its output on the current output line with one space between values. For information about using these functions, see Sections 5.8.5.1 through 5.8.5.3.

An example of a right-hand-side expression follows:

```
(CRLF) (CRLF) |There are| <VALUE> |checks dated|
              <DAY> <MONTH> <YEAR> (CRLF)
```

NOTE

Specify the right-hand-side expression as a list of atoms. If you specify attributes and their values in pairs (for example, ^NUMBER 102), the order of the output depends on the assignment of fields to attribute names.

Example

Consider the following WRITE action:

```
(WRITE (CRLF) (CRLF) |There are| <VALUE> |checks dated |
                     <DAY> <MONTH> <YEAR> (CRLF))
```

This action produces the following output if the variable <VALUE> is bound to 5 and the variables <DAY>, <MONTH>, and <YEAR> are bound to 14, NOV, and 1988, respectively:

```
There are 5 checks dated 14 NOV 1988
```

Chapter 11

Functions

This chapter provides descriptions of the VAX OPS5 functions. A function is a subroutine that performs an operation and returns one or more values to the program.

Table 11–1 lists the names of the functions and provides brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name. For information about function calls, see Section 3.2.1.6.

Most VAX OPS5 functions require at least one argument. You can represent argument values with atoms, variables bound to atoms, or calls to functions that return a single atom (that is, COMPUTE, GENATOM, or external functions). When you specify argument values, separate them with any combination of spaces, tabs, and carriage returns.

Table 11–1: Summary of Functions

Function	Description
ACCEPT	Reads an atom or a list of atoms (enclosed in parentheses) from the terminal or a file
ACCEPTLINE	Reads a line of input consisting of one or more atoms and lists of atoms (enclosed in parentheses) from the terminal or a file
COMPUTE	Evaluates an arithmetic expression and returns the result
CRLF	Causes the WRITE action to produce output on a new line
GENATOM	Returns a system-generated atom
LITVAL	Returns the integer that represents an attribute's field
RJUST	Causes the WRITE action to right justify output in a field of a specified width
SUBSTR	Copies a sequence of atoms from a working-memory element to output produced by the WRITE action or to another working-memory element created with the MAKE action or modified by the MODIFY action.
TABTO	Causes the WRITE action to start writing output in a specified column

ACCEPT

Reads an atom or a list of atoms (enclosed in parentheses) from the terminal or a file. The list can contain parentheses; the function continues reading until it finds a closing parenthesis that matches the first opening parenthesis, then it discards the outermost pair of parentheses, but returns all other parentheses as atoms. If the function reads only "( )", it returns the value NIL.

By default, the ACCEPT function reads input from the terminal. If you want the function to read input from a file, call the function with the file identifier of an open input file, or change the default for input, using the DEFAULT action (see Section 5.8.2).

When the ACCEPT function reads past the end of a file, the function returns the symbol END–OF–FILE.

Format

ACCEPT *[file-id]*

Argument

file-id
The file identifier of the file from which input is to be read. This argument is optional. If you do not specify a file identifier, or if the name you specify is not associated with an open input file, input is read from the current default for the ACCEPT function (set with the DEFAULT action or command).

Example

The MAKE action in the following production uses the ACCEPT function to read input:

```
(P WHAT-NUMBER
   { <START>
     (START) }
  -->
   (REMOVE <START>)
   (WRITE (CRLF) |What number are you looking for? |)
   (MAKE REPLY ^NUMBER (ACCEPT)))
```

The attribute ^NUMBER is given the value read by the ACCEPT function.

ACCEPTLINE

Reads a line of input terminated by a carriage-return character, discarding all unquoted parentheses.

If some of the atoms on a line of input have already been read, the line of input is defined as all the remaining atoms on the current line, that is, from the current position to the next carriage return (CRLF) character; if no atoms have been read from the current line, the input line is defined as the whole of the current line; if all the atoms on the current line have been read, the input line is the line following the current line.

If the input line contains no atoms, the default values are used.

By default, the ACCEPTLINE function reads input from the terminal. If you want the function to read input from a file, call the function with the file identifier of an open input file or change the default for input, using the DEFAULT action (see Section 5.8.2).

Format

ACCEPTLINE *[file-id] [default-value...]*

Arguments

file-id
The file identifier of the file from which input is to be read. This argument is optional. If you do not specify a file identifier, or if the name you specify is not associated with an open input file, input is read from the current default for the ACCEPTLINE function (set with the DEFAULT action or command).

default-value
An atom that the ACCEPTLINE function is to return if the function reads:

- A line of input from the terminal that consists only of a carriage return
- A file that consists of a line of spaces and tabs
- Past the end of a file

This argument is optional. If you do not specify a default value, the function does not return a value in these cases.

You can specify one or more default values.

Example

The MAKE action in the following production contains a call to the ACCEPTLINE function, which returns the atoms NO and DATE if the input line is empty:

```
(P WHAT-DATE
   { <START>
     (START) }
  -->
   (REMOVE <START>)
   (WRITE (CRLF) |What date do you want to search for? |)
   (MAKE REPLY ^DATE (ACCEPTLINE NO DATE)))
```

The function call reads the values you type at the terminal until you press the Return key. The MAKE action uses the values to create a working-memory element that has the class name REPLY and the attribute ^DATE. If the default for input is a file, the values are read from that file rather than from the terminal. If the function reads a blank line from the file, or if input is from the terminal and the user only presses the Return key, the function returns the atoms NO and DATE to the attribute ^DATE.

COMPUTE

Evaluates an arithmetic expression and returns the result.

Format

COMPUTE *arithmetic-expression*

Argument

arithmetic-expression
The arithmetic expression to be evaluated. An arithmetic expression can contain numbers, variables bound to numbers, arithmetic operators, and function calls. If an expression contains both an integer and a floating-point number, the result is a floating-point number.

NOTE

If you specify a function, it must return only one atom in register R0.

The operators you can use are:

+	Addition
–	Subtraction
*	Multiplication
//	Division
\\	Modulus

NOTE

Use the modulus operator only with an integer or a variable bound to an integer.

Use infix notation in VAX OPS5 arithmetic expressions; that is, place operators between operands. Separate each operator and operand with a space.

All operators have the same priority, and the COMPUTE function evaluates them from right to left. To override the right-to-left evaluation, use parentheses.

Example

Consider the following MODIFY action:

```
(MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>))
```

The call to the COMPUTE function adds 1 to the value bound to the variable <VALUE> and returns the result to the attribute ^VALUE.

CRLF

Causes the WRITE action to produce output on a new line.

NOTE

Use the CRLF function with the WRITE action only.

Format

CRLF

Example

Consider the following production:

```
(P WHAT-DATE
     { <START>
       (START) }
   -->
     (REMOVE <START>)
     (WRITE (CRLF) (CRLF) |What date do you want to search for? |
            (CRLF) (CRLF) |Enter the day, the first three|
                          |letters of the month, and the year.|
                   (CRLF)
                          |For example -- 14 NOV 1988|
            (CRLF) (CRLF)
                          |Type STOP to halt the program.|
            (CRLF) (CRLF)
                          |Date>>> |)
     (MAKE COUNT ^VALUE 0)
     (MAKE REPLY ^DATE (ACCEPTLINE)))
```

The WRITE action produces the following output:

```
What date do you want to search for?

Enter the day, the first three letters of the month, and the year.
For example -- 14 NOV 1988

Type STOP to halt the program.

Date>>>
```

GENATOM

Returns a system-generated atom. Each time the run-time system starts executing a program, the first call to the GENATOM function generates the atom G:1. The second atom the function generates is G:2, the third atom is G:3, and so on. For information about how to use system-generated atoms, see Section 5.10.

Format

GENATOM

Example

The following BIND action binds the variable <TRANSACTION–ID> to the atom produced by the GENATOM function:

```
(BIND <TRANSACTION-ID> (GENATOM))
```

LITVAL

Returns the integer that represents an attribute's field. Use this function to compute the field that contains an atom in a vector attribute's value. For information about how to bind a variable to an attribute's field, see Section 5.6.2.

Format

LITVAL *attribute-name*

Argument

attribute-name
The name of an attribute.

Example

Suppose a LITERALIZE declaration assigned field 2 to the attribute name NUMBER. Consider the following WRITE action:

```
(WRITE |Field| (LITVAL NUMBER)
       |is assigned to the attribute ^NUMBER.|)
```

The call to the LITVAL function returns the atom 2, and the WRITE action displays the following output:

```
Field 2 is assigned to the attribute ^NUMBER.
```

RJUST

Causes the WRITE action to right-justify output in a field of a specified width. This function is useful for writing a column of numbers with decimal positions aligned.

Calls to the RJUST function can follow calls to the CRLF and TABTO functions but must directly precede the value being written. For example:

```
(WRITE (CRLF)
       (TABTO 5)
       (RJUST 10) |250.00|)
```

NOTE

Use the RJUST function with the WRITE action only.

Format

RJUST *width*

Argument

width
An integer that indicates the width of the field in which output is to be placed. If the output being written requires more character positions than you specify for the field, the WRITE action writes the output as if the RJUST function had not been specified. That is, the action inserts one space and then writes the output.

Example

The following WRITE action writes a vertical list of numbers right-justified in a column 10 characters wide:

```
(WRITE (CRLF) (RJUST 10) |10.06|
       (CRLF) (RJUST 10) |2.45|
       (CRLF) (RJUST 10) |56.00|
       (CRLF) (RJUST 10) |250.00|)
```

The output is:

```
     10.06
      2.45
     56.00
    250.00
```

SUBSTR

Copies a sequence of atoms from a working-memory element to output produced by the WRITE action or to another working-memory element created with the MAKE action or modified by the MODIFY action.

Format

SUBSTR *element-designator first-value last-value*

Arguments

element-designator
An element variable or integer that refers to a condition element on the left-hand side of the production, which indicates the working-memory element from which atoms are to be copied.

first-value
An attribute name or integer that refers to the first atom to be copied from the working-memory element.

last-value
An attribute name or integer that refers to the last atom to be copied from the working-memory element. You can also specify the symbol INF, which causes the function to copy atoms until it gets to the end of the element. If the first and last values that you specify are the same, the function copies only one atom.

Example

The WRITE action in the following production contains a call to the SUBSTR function:

```
(P FIND-CHECKS
   { <REPLY>
     (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
   { <CHECK>
     (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
            ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
   { <COUNTER>
     (COUNT ^VALUE <VALUE>) }
  -->
   (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                        |for $| <AMOUNT>
                        |dated| (SUBSTR <REPLY> DATE INF))
   (MODIFY <CHECK> ^COUNTED YES)
   (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The SUBSTR function copies the atoms in the value of the vector attribute ^DATE. The WRITE action then includes those atoms in its output.

TABTO

Causes the WRITE action to start writing output in a specified column.

NOTE

Use the TABTO function with the WRITE action only.

Format

TABTO *column*

Argument

column
An integer that indicates the column in which the WRITE action is to start writing output. If you specify a column that is to the left of the last column in which output is written, the WRITE action writes the output on a new line, starting at the specified column.

Example

The following WRITE action displays the headers of three columns:

```
(WRITE (CRLF) (TABTO 10) NUMBER
              (TABTO 25) AMOUNT
              (TABTO 40) DATE)
```

The output is:

```
NUMBER          AMOUNT          DATE
```

Chapter 12

Command Interpreter Commands

This chapter describes the commands you can use with the VAX OPS5 command interpreter. Table 12–1 lists the names of the commands and gives brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name. For more information about commands, see the *VAX OPS5 User's Guide*.

Most VAX OPS5 commands require at least one argument. Argument values must be atoms; they cannot be variables or function calls and cannot include the quote operator (//). When you specify argument values, separate them with any combination of spaces, tabs, and carriage returns.

There are two ways of entering commands: with and without enclosing parentheses. If you start without an opening parenthesis, the command interpreter does not check pairing of parentheses within the command. You can continue the command on a new line by ending the line with a hyphen (-) and a carriage return.

If you start a command with an opening parenthesis, every opening parenthesis must have a corresponding closing one. If you want to include an extra parenthesis in a command you have started this way, you must include it in quote characters as follows: | (| , or the interpreter will not allow you to end the command until it finds the corresponding parenthesis. Also, you can continue the command over any number of lines, ending each line with a carriage return and starting each new line at the continuation prompt (_OPS5>).

Nesting parentheses is convenient for entering the body of a production for use by a BUILD action. For example:

```
OPS5>(MAKE BUILDME NEW-PRODUCTION (<X>) --> (WRITE|Hi|))
```

You can abbreviate commands, as long as the abbreviation is unambiguous.

The command interpreter treats a semicolon (;) as a comment character, allowing you to include comments in your commands. This is particularly useful in command files that are executed with the @ command. If a command argument includes a semicolon, enclose the argument in vertical bars (| |) to ensure that the argument is evaluated and not treated as a comment.

Table 12–1: Summary of Commands

Command	Description
@	Opens a file containing VAX OPS5 commands and executes the commands
ADDSTATE	Adds the contents of a file produced by the SAVESTATE action or command to the current state of working memory and the conflict set
AFTER	Specifies the number of recognize-act cycles that must be executed before a specified catcher is executed
BACK	Restores working memory and the conflict set to the state of a previous recognize-act cycle
BUILD	Adds a STARTUP statement, a production, or a catcher to a halted executable image
CALL	Calls an external subroutine
CLOSEFILE	Closes the open files associated with specified file identifiers and dissociates the identifiers from the files
CS	Displays the current contents of the conflict set
DEFAULT	Sets the terminal or a file as the default input source for the ACCEPT and ACCEPTLINE functions, or the default output destination for the WRITE action or trace output
DISABLE	Disables run-time system features
ENABLE	Enables run-time system features
EXCISE	Disables productions
EXIT	Exits from the command interpreter and returns control to the operating system
MAKE	Creates a working-memory element
MATCHES	Displays the time tags of working-memory elements that match condition elements in specified productions
MODIFY	Changes one or more atoms in an existing working-memory element
NEXT	Displays the instantiation the run-time system will select from the conflict set for the act phase of the next recognize-act cycle
OPENFILE	Opens a file and associates it with a file identifier
PBREAK	Displays productions that have breakpoints set, sets breakpoints for productions, or deletes breakpoints from productions
PPWM	Displays working-memory elements that match a specified element pattern
REMOVE	Deletes elements from working memory
REPORT	Generates a timing report and a cause report that you can use to debug and optimize VAX OPS5 programs
RESTART	Reruns an OPS5 program from the OPS5 level
RESTORESTATE	Clears and then restores working memory and the conflict set to the state recorded in a file produced by the SAVESTATE action or command
RUN	Executes recognize-act cycles
SAVESTATE	Copies the state of working memory and the conflict set to a file
SHOW SPACE	Displays information about working memory and the VAX OPS5 symbol table

(continued on next page)

Table 12–1 (Cont.): Summary of Commands

Command	Description
STRATEGY	Displays or sets the conflict-resolution strategy
WATCH	Displays or sets the run-time system's trace level
WBREAK	Displays working-memory elements that have breakpoints set, sets breakpoints for working-memory elements, or deletes breakpoints from working-memory elements
WM	Displays working-memory elements

@

Opens a file containing VAX OPS5 commands and executes the commands. The file must contain only VAX OPS5 commands. If the file cannot be opened, the run-time system displays the following message:

```
%OPSRT-W-OPENERR, @ -- unable to open specified file
```

You can use the @ command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

Format

@ file-spec

Argument

file-spec
The VMS file specification for a file containing VAX OPS5 commands to be executed.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Example

Suppose the file CHECKS.DAT consists of the following commands:

```
(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO
            ^DATE 2 NOV 1988)
(MAKE CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO
            ^DATE 14 NOV 1988)
(MAKE CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO
            ^DATE 14 NOV 1988)
(MAKE CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO
            ^DATE 25 NOV 1988)
(MAKE START)
(STRATEGY MEA)
```

The following command opens the file CHECKS.DAT and executes five MAKE commands and a STRATEGY command:

```
OPS5>@ CHECKS.DAT
```

ADDSTATE

Adds the contents of a file produced by the SAVESTATE action or command to the current state of working memory and the conflict set. The time tags of the working memory-elements in the saved file are ignored.

Format

ADDSTATE *file-spec*

Argument

file-spec
A VMS file specification for a file previously produced by the SAVESTATE action or command.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Example

Suppose you use the SAVESTATE command to store the state of working memory and the conflict set in the file CHECKS.DAT. The following command adds the contents of the file CHECKS.DAT to the current state of working memory and the conflict set:

```
OPS5>ADDSTATE CHECKS.DAT
```

AFTER

Specifies the number of recognize-act cycles that must be executed before a specified catcher is executed, thus controlling loops in a program.

For more information about catchers and controlling loops, see Section 5.9.

Format

AFTER *cycles catcher-name*

Arguments

cycles
A positive integer that specifies the number of recognize-act cycles that are to be executed before the specified catcher is executed. If execution halts before the specified number of cycles has been executed, the catcher is not executed.

catcher-name
A symbol that names a catcher.

Example

The following command specifies that the catcher named BALANCE is to be executed after five recognize-act cycles have been executed:

```
OPS5>AFTER 5 BALANCE
```

If program execution halts before five recognize-act cycles have been executed, the catcher is not executed.

BACK

Restores working memory and the conflict set to the state of a previous recognize-act cycle. However, the command does not restore the effects of actions and commands that do not modify working memory or the conflict set (such as I/O operations).

By default, the BACK command is disabled to increase the speed of program execution. To enable the command for debugging, use the ENABLE command with the keyword BACK (see the description of the ENABLE command).

For more information about backing up over recognize-act cycles, see the *VAX OPS5 User's Guide*.

Format

BACK *[cycles]*

Argument

cycles
An integer that specifies the number of recognize-act cycles the run-time system is to back up. The maximum number of cycles that can be backed up is 64.

This argument is optional. If you do not specify the argument, the system backs up one cycle.

Example

The following command restores working memory and the conflict set to the state that existed before the last three recognize-act cycles were executed:

```
OPS5>BACK 3
```

BUILD

Adds a STARTUP statement, a production, or a catcher to a running program that has been paused.

Use the following procedure.

1. Enter BUILD at the OPS5> prompt. OPS5 should display the _BUILD> prompt.
2. Enter the information you want to add to the image. Use as many lines as necessary.
3. Enter ENDBUILD or type Ctrl/Z. OPS5 should display the OPS5> prompt.
4. Enter RESTART to resume execution with the new information in effect.

Format

BUILD

Example

TEST.EXE is a program that does not yet contain any constructs. The following entries add a STARTUP statement to TEST.EXE.

```
OPS5>BUILD
_BUILD> (STARTUP (MAKE X))
_BUILD> ENDBUILD
OPS5>RESTART
```

The following entries add a production to TEST.EXE.

```
OPS5>BUILD
_BUILD> (P TEST (<X>) --> (WRITE (CRLF) |Dear Subscriber:|))
_BUILD> Ctrl/Z
OPS5>RESTART
```

Now TEST.EXE prints the message:

```
Dear Subscriber:
```

CALL

Calls an external subroutine, which is a routine written in a VAX language other than VAX OPS5.

For information about calling external subroutines, see Chapter 6.

Format

CALL *external-routine-name [external-routine-argument...]*

Arguments

external-routine-name
The name of the external subroutine to be called. Specify the name of a subroutine that has been declared with the EXTERNAL declaration. Otherwise, the run-time system displays the following warning:

```
?OPSRT-W-NOTEXTERNAL, CALL -- routine not declared external:
AAAAAA
```

external-routine-argument
A number or symbol representing the value of an external routine's argument. The arguments are placed in order in the result element, starting in position one, and can be retrieved by using the OPS$PARAMETER support routine.

Example

The following EXTERNAL declaration declares an external subroutine named READ_NYSE_EXTRACT:

```
(EXTERNAL READ_NYSE_EXTRACT)
```

The following command calls the subroutine READ_NYSE_EXTRACT with the atom DISNEY as an argument:

```
OPS5>CALL READ_NYSE_EXTRACT DISNEY
```

CLOSEFILE

Closes the open files associated with specified file identifiers and dissociates the identifiers from the files.

Format

CLOSEFILE *file-id...*

Argument

file-id
The file identifier of an open file to be closed. You can specify one or more file identifiers.

Example

The following command closes the open files associated with the file identifiers CHECKSI and CHECKSO:

```
OPS5>CLOSEFILE CHECKSI CHECKSO
```

CS

Displays the current contents of the conflict set. This contains instantiations that include a production name and a list of time tags of working-memory elements that satisfy that production's left-hand side. The CS command displays instantiations in the following format:

production time-tag–1 time-tag–2...

where time-tag–1 is the time tag of a working-memory element that matches the first condition element on the left-hand side, time-tag–2 matches the second, and so on.

Format

CS

Example

The following command displays the contents of the conflict set:

```
OPS5>CS
FIND-CHECKS 12 3 11
FIND-CHECKS 12 4 11
FIND-CHECKS 12 5 11
FIND-CHECKS 12 6 11
FIND-CHECKS 12 2 11
```

DEFAULT

Sets the terminal or a file as the default input source for the ACCEPT and ACCEPTLINE functions, or the default output destination for the WRITE action or trace output. If you do not use the DEFAULT command or action to specify otherwise, the default source for input and destination for output are SYS$INPUT and SYS$OUTPUT.

Format

DEFAULT *location keyword*

Arguments

location
The source from which input is to be read or the destination to which output is to be written. The value can be either a file identifier or the symbol NIL. If you specify a file identifier, the DEFAULT command sets the source or destination to the open file associated with that name. If you specify NIL, the input is read from or output is sent to the terminal.

keyword
A keyword that specifies whether the default is to be set for the ACCEPT or ACCEPTLINE functions, the WRITE action, or trace output. Table 12–2 lists the keywords you can specify.

Table 12–2: DEFAULT Command Keywords

Keyword	Description
ACCEPT	Input read by the ACCEPT and ACCEPTLINE functions is read from the specified source.
TRACE	Trace output is sent to the specified destination. You can enable and disable trace output, using the WATCH command.
WRITE	Output produced by the WRITE action is sent to the specified destination.

Example

The following command sets the open file associated with the file identifier name CHECKSI to be the default source of input for the ACCEPT and ACCEPTLINE functions:

```
OPS5>DEFAULT CHECKSI ACCEPT
```

To set the default back to the terminal, specify:

```
OPS5>DEFAULT NIL ACCEPT
```

DISABLE

Disables run-time system features, such as the run-time system message display, the Performance Measurement and Evaluation (PME) package, and the BACK command. To disable a feature, specify the appropriate keyword.

You can use the DISABLE command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

For more information about disabling run-time system features, see the *VAX OPS5 User's Guide.*

Format

DISABLE *keyword*

Argument

keyword
A keyword that specifies the feature to be disabled. The keywords and the facilities they disable are listed in Table 12–3.

Table 12–3: DISABLE Command Keywords

Keyword	Facility	Default
BACK	BACK command	Disabled
HALT	Run-time informational messages	Enabled
TIMING	PME package	Disabled
WARNING	Run-time warning and fatal error messages	Enabled

Example

The following command disables the BACK command:

```
OPS5>DISABLE BACK
```

ENABLE

Enables run-time system features, such as the run-time system message display, the Performance Measurement and Evaluation (PME) package, and the BACK command. To enable a feature, specify the appropriate keyword.

You can use the ENABLE command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

For more information about enabling run-time system features, see the *VAX OPS5 User's Guide*.

Format

ENABLE *keyword*

Argument

keyword
A keyword that specifies the feature to be enabled. The keywords and the facilities they enable are listed in Table 12–4.

Table 12–4: ENABLE Command Keywords

Keyword	Facility	Default
BACK	BACK command	Disabled
HALT	Run-time informational messages	Enabled
TIMING	PME package	Disabled
WARNING	Run-time warning and fatal error messages	Enabled

Example

The following command enables the BACK command:

```
OPS5>ENABLE BACK
```

EXCISE

Disables productions. After a production has been disabled, it cannot be executed. To use the production again, you must exit from the command interpreter and reexecute the program.

Use the EXCISE command to disable productions that appear to be causing errors.

Format

EXCISE *production-name...*

Argument

production-name
The name of a production to be disabled. You can specify the name of one or more productions.

Example

The following command disables the productions named FIND–CHECKS and COUNTED–CHECKS:

```
OPS5>EXCISE FIND-CHECKS COUNTED-CHECKS
```

EXIT

Exits from the command interpreter and returns control to the operating system, or to the calling program if the VAX OPS5 program was called as a subroutine. Using this command is equivalent to typing Ctrl/Z.

Format

EXIT

Example

The following command exits from the command interpreter and returns control to the operating system:

```
OPS5>EXIT
$
```

MAKE

Creates a working-memory element.

For more information about working-memory elements, see Section 2.1.

Format

MAKE *[class-name] [{scalar-attribute value}...] [vector-attribute value]*

Arguments

class-name
A symbol that names the class of the element to be created. This argument is optional. If you specify a class name, the command places the name in the first field of the working-memory element. If you do not specify a class name, the command places NIL in the first field of the element.

scalar-attribute
A scalar attribute that describes a characteristic of the working-memory element to be created and specifies the field of the working-memory element in which the corresponding value will be placed. You must specify each scalar attribute with a value.

value
An atom to be placed in the field of the working-memory element indicated by the corresponding attribute.

vector-attribute
A vector attribute that describes a characteristic of the working-memory element to be created. You must specify a vector attribute with a value. Specify only one vector attribute in a MAKE command.

value
One or more atoms to be placed in the working-memory element. The first atom is placed in the field indicated by the vector attribute. If the vector attribute is declared, the first atom is placed in field 256, and the remaining atoms are placed sequentially in the fields that follow.

Example

Consider the following MAKE command:

```
OPS5>(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
```

This command creates the following working-memory element:

```
1 [NIL] (CHECK) ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
```

MATCHES

Displays the time tags of working-memory elements that match condition elements in specified productions. The command lists the time tags for the working-memory elements that match the first condition element, then the second condition element, and so on. The command lists the time tags as follows:

```
*** matches for n ***
time-tag
   .
   .
   .
```

The n in the preceding output indicates the position of the condition element in the production. For example, if the condition element is the first element in a production, n is 1.

Working-memory elements might match more than one condition element, particularly if the condition elements in productions contain variables. Therefore, the MATCHES command also displays the time tags of the working-memory elements that match more than one condition element.

For more information about displaying match information, see the *VAX OPS5 User's Guide*.

Format

MATCHES *production-name...*

Argument

production-name
The name of a production for which match information is to be displayed. You can specify the name of one or more productions.

Example

Suppose working memory contains the following elements:

```
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6 [WHAT-DATE](COUNT ^VALUE 0)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

Consider the following production:

```
(P FIND-CHECKS
   { <REPLY>
     (REPLY ^DATE { <DAY> <> STOP } <MONTH> <YEAR>) }
   { <CHECK>
     (CHECK ^NUMBER <NUMBER> ^AMOUNT <AMOUNT>
            ^COUNTED NO ^DATE <DAY> <MONTH> <YEAR>) }
   { <COUNTER>
     (COUNT ^VALUE <VALUE>) }
  -->
   (WRITE (CRLF) (CRLF) |Found check number| <NUMBER>
                        |for $| <AMOUNT>
                        |dated| (SUBSTR <REPLY> DATE INF))
   (MODIFY <CHECK> ^COUNTED YES)
   (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The following command displays matches for this production:

```
OPS5>MATCHES FIND-CHECKS
>>> FIND-CHECKS <<<
*** matches for 1 ***
7
*** matches for 2 ***
1
2
3
4
*** matches for 1 2 ***
7 2
7 3
*** matches for 3 ***
6
```

MODIFY

Changes one or more atoms in an existing working-memory element. The command removes an element from working memory and uses the atoms in that element and new atoms you specify to create a new element. Therefore, when you modify a working-memory element, the element's time tag changes.

You can change more than one atom in a MODIFY command.

For more information about working-memory elements, see Section 2.1.

Format

MODIFY *time-tag {attribute value}...*

Arguments

time-tag
The time tag of the working-memory element whose atoms are to be changed.

attribute
An attribute that specifies which value in the working-memory element is to be changed. You must specify each attribute with a value.

value
An atom (or a list of atoms if you specify a vector attribute) to replace one or more atoms in the working-memory element. The corresponding attribute indicates the field in which the new atom is placed.

Example

Suppose working memory contains the following elements:

```
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6 [WHAT-DATE](COUNT ^VALUE 0)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

The following command changes an atom in the working-memory element whose time tag is 3:

```
OPS5>MODIFY 3 ^COUNTED YES
```

The command removes the element whose time tag is 3 from working memory and creates the following working-memory element:

```
9 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED YES ^DATE 14 NOV 1988)
```

NEXT

Displays the instantiation the run-time system will select from the conflict set for the act phase of the next recognize-act cycle. The NEXT command displays the instantiation in the following format:

production time-tag–1 time-tag–2...

The output format is the same as for the CS command.

Format

NEXT

Example

The following command shows that the next instantiation the run-time system will select from the conflict set is FIND–CHECKS:

```
OPS5>NEXT
FIND-CHECKS 12 6 11
```

OPENFILE

Opens a file and associates it with a file identifier.

Format

OPENFILE *file-id file-spec keyword*

Arguments

file-id
A symbol that represents the file identifier with which the specified file is to be associated.

file-spec
The VMS file specification for the file to be opened. If you are opening a file for input, the file must already exist.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

keyword
A keyword that indicates whether the specified file is to be opened for input or output. If you specify IN, the command opens an existing file for reading only. If you specify OUT, the command creates a new file and opens it for writing only. If you specify APPEND, the command opens an existing file for writing, and sets the file pointer to the end of the file.

Example

The following command opens the file CHECKS.DAT for input and associates it with the file identifier CHECKSI:

```
OPS5>OPENFILE CHECKSI CHECKS.DAT IN
```

PBREAK

Displays productions that have breakpoints set, sets breakpoints for productions, or deletes breakpoints from productions. When the run-time system encounters a breakpoint, the system finishes executing the current recognize-act cycle, displays the following message, and invokes the command interpreter:

```
%OPSRT-I-PBREAK, PBREAK encountered

OPS5>
```

Format

PBREAK *[production-name...]*

Argument

production-name
The name of a production. You can specify one or more production names. If you specify the name of a production for which a breakpoint is not set, the command sets a breakpoint. If you specify the name of a production for which a breakpoint is set, the command deletes the breakpoint.

The argument is optional. If you do not specify the name of a production, the command displays the names of the productions for which breakpoints are set.

Example

The following command displays the names of the productions for which breakpoints are set:

```
OPS5>PBREAK
WHAT-DATE
COUNTED-CHECKS
```

Suppose a breakpoint is not set for the production named FIND–CHECKS, and a breakpoint is set for the production named COUNTED–CHECKS. The following commands set a breakpoint for the production FIND–CHECKS and delete the breakpoint from the production COUNTED–CHECKS:

```
OPS5>PBREAK
COUNTED-CHECKS
OPS5>PBREAK FIND-CHECKS COUNTED-CHECKS
OPS5>PBREAK
FIND-CHECKS
```

PPWM

Displays working-memory elements that match a specified element pattern. Element patterns are similar to working-memory elements and can include a class name, scalar attributes specified with values, and a vector attribute specified with a value.

When the command displays a working-memory element, the output includes the following information:

- Time tag
- Name of the production that created the element
- List of atoms in the element

The system displays this information in the following format:

time-tag [production-name] (class-name attribute-1 value-1 attribute-2 value-2...)

If you do not specify a pattern, the command displays all the elements in working memory.

Format

PPWM *[class-name] [{scalar-attribute value}...] [{vector-attribute value}]*

Arguments

class-name
A symbol that names the class of working-memory elements to be displayed. This argument is optional.

scalar-attribute
A scalar attribute that describes a characteristic of the working-memory elements to be displayed and specifies the field of the working-memory elements in which the corresponding value is stored. You must specify each scalar attribute with a value.

value
An atom in the working-memory elements to be displayed.

vector-attribute
A vector attribute that describes a characteristic of the working-memory elements to be displayed. You must specify a vector attribute with a value. Specify only one vector attribute in an element pattern.

value
One or more atoms in the working-memory elements to be displayed.

Example

The following command displays all elements in working memory:

```
OPS5>PPWM
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6 [WHAT-DATE](COUNT ^VALUE 0)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

The following command displays the elements in working memory that match the specified element pattern:

```
OPS5>PPWM ^DATE 14 NOV 1988
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

REMOVE

Deletes elements from working memory.

Format

REMOVE *time-tag...*

Argument

time-tag
An integer that represents the time tag of a working-memory element to be deleted. You can specify one or more time tags. You can also specify an asterisk (*) to delete all working-memory elements.

Example

The following command deletes all working-memory elements:

```
OPS5>REMOVE *
```

The following command deletes the working-memory elements whose time tags are 3 and 4:

```
OPS5>REMOVE 3 4
```

REPORT

Generates a timing report and a cause report that you can use to debug and optimize VAX OPS5 programs. The reports include the output of the Performance Measurement and Evaluation (PME) package and are placed in the files TIMINGCPU.TXT and TIMINGCAU.TXT. When you exit from the VAX OPS5 command interpreter, you can use the DCL command TYPE to display the reports on the terminal or the DCL command PRINT to print hard-copy listings of the reports.

For more information about the PME package, see the *VAX OPS5 User's Guide*.

Format

REPORT *keyword*

Argument

keyword
The keyword TIMING causes the REPORT command to generate reports using the data collected by the PME package. The PME package is disabled by default. To enable and disable the package, use the ENABLE and DISABLE commands with the keyword TIMING.

Example

The following command generates a timing and cause report, using the output of the PME package, and places the reports in the files TIMINGCPU.TXT and TIMINGCAU.TXT:

```
OPS5>REPORT TIMING
```

RESTART

Lets you rerun a paused VAX OPS5 program from the beginning, without exiting to the DCL level.

The RESTART command:

- Removes all elements from working memory and the conflict set
- Resets the time-tag counter
- Resets the DEFAULT WRITE, DEFAULT ACCEPT, and DEFAULT TRACE files to NIL
- Closes all files opened with the OPENFILE command
- Resets the recognize-act cycle counter
- Executes the STARTUP statement

Format

RESTART

Example

The following command reruns from the beginning a program that has been paused.

```
OPS5>RESTART
```

RESTORESTATE

Clears and then restores working memory and the conflict set to the state recorded in a file produced by the SAVESTATE action or command.

Format

RESTORESTATE *file-spec*

Argument

file-spec
The VMS file specification for a file previously produced by the SAVESTATE action or command. The command uses the contents of the file to restore working memory and the conflict set.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Example

The following command clears and then restores the contents of working memory and the conflict set to the same state recorded in the file CHECKS.DAT:

```
OPS5>RESTORESTATE CHECKS.DAT
```

RUN

Causes the run-time system to execute recognize-act cycles. The run-time system does not execute recognize-act cycles until this command has been executed. You can use the RUN command to control the number of recognize-act cycles the system executes.

You can use the RUN command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

For more information about executing recognize-act cycles, see the *VAX OPS5 User's Guide*.

Format

RUN *[integer]*

Argument

integer
The number of recognize-act cycles the run-time system is to execute. This argument is optional. If you do not specify an integer, the run-time system executes recognize-act cycles until no productions are satisfied or a HALT action, breakpoint, or Ctrl/C interrupts execution.

NOTE

If the program halts, the run-time system does not execute the number of recognize-act cycles indicated by the integer.

Example

The following command starts executing recognize-act cycles:

```
OPS5>RUN
```

The following command executes four recognize-act cycles:

```
OPS5>RUN 4
```

The following STARTUP statement includes the RUN command:

```
(STARTUP
        (RUN))
```

When you enter the DCL command RUN to execute a program that includes this statement, the run-time system starts executing recognize-act cycles without invoking the command interpreter.

SAVESTATE

Copies the state of working memory and the conflict set to a file. You can restore the contents of that file to working memory and the conflict set later, using the ADDSTATE and RESTORESTATE actions and commands.

Format

SAVESTATE *file-spec*

Argument

file-spec
The VMS file specification for the file to which the command is to copy the state of working memory and the conflict set.

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Example

The following command copies the state of working memory and the conflict set to the file CHECKS.DAT:

```
OPS5>SAVESTATE CHECKS.DAT
```

SHOW SPACE

Displays information about working memory and the VAX OPS5 symbol table. The information consists of the current number of working-memory elements and symbol-table entries, the amount of memory that these currently occupy, and the largest value that these have reached during this execution of the program.

Format

SHOW SPACE

Example

```
OPS5>SHOW SPACE
  WORKING-MEMORY ELEMENTS             SYMBOL TABLE ENTRIES
  Current number      -- 10           Current number      -- 97
  Current space used --   0.6 KBytes  Current space used --   4.4 KBytes
  Maximum number      -- 12           Maximum number      -- 97
  Maximum space used --   0.7 KBytes  Maximum space used --   4.4 KBytes
```

STRATEGY

Displays or sets the conflict-resolution strategy, which can be either means-ends-analysis (MEA) or lexicographic-sort (LEX). The default is the LEX strategy.

You can use the STRATEGY command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

Format

STRATEGY *[keyword]*

Argument

keyword
A keyword that specifies the conflict-resolution strategy to be set. Specify the keyword LEX to enable the LEX strategy or the keyword MEA to enable the MEA strategy.

This argument is optional. If you do not specify a keyword, the command displays the strategy that is enabled.

Example

The following command shows that the LEX strategy is enabled:

```
OPS5>STRATEGY
LEX
```

The following command enables the MEA strategy:

```
OPS5>STRATEGY MEA
```

WATCH

Displays or sets the run-time system's trace level. The amount of trace information the system displays while executing a program depends on the system's current trace level. The trace levels are represented with the integers 0 to 4, or with the names RULE, NORULE, WM, NOWM, CS, NOCS, PM, NOPM, ALL, and NOALL. Each level is described in Table 12–5.

You can use the WATCH command in a STARTUP statement. For a description of the STARTUP statement, see Chapter 9.

For details about trace output, see the *VAX OPS5 User's Guide*.

Format

$$\textbf{WATCH}\left\{\begin{array}{l}\textit{[trace-level]}\\ \textit{[trace-name]...}\end{array}\right\}$$

Arguments

trace-level
An integer in the range 0 to 4 that represents the trace level to be set. You can specify only one trace level in a WATCH command.

trace-name
The name of the trace level, which is added to the level currently set. You can give a list of trace names, in which case these are applied from left to right.

Table 12–5 lists the trace levels and describes their effects.

Table 12–5: Trace Levels

Level	Effect
RULE	Enables tracing of firing rules
NORULE	Disables rule tracing
WM	Enables tracing of working-memory elements into and out of working memory
NOWM	Disables working-memory tracing
CS	Enables tracing of instantiations into and out of the conflict set
NOCS	Disables conflict set tracing
PM	Enables tracing of productions into and out of production memory
NOPM	Disables production-memory tracing
ALL	Enables RULE, WM, CS, and PM tracing
NOALL	Disables all tracing
0	Disables all tracing

(continued on next page)

Table 12–5 (Cont.): Trace Levels

Level	Effect
1	Enables RULE tracing
2	Enables RULE and WM tracing
3	Enables RULE, WM, and CS tracing
4	Enables RULE, WM, CS, and PM tracing

The argument is optional. If you do not specify a trace level, the command displays the run-time system's current trace level.

Example

```
OPS5>WATCH 2
OPS5>WATCH
RULE WM
OPS5>WATCH CS NORULE
OPS5>WATCH
WM CS
```

WBREAK

Displays working-memory elements that have breakpoints set, sets breakpoints for working-memory elements, or deletes breakpoints from working-memory elements.

If a breakpoint is set for a working-memory element, the run-time system encounters that breakpoint after executing the right-hand side of the production that created the element. At that point, the system finishes executing the current recognize-act cycle, displays the following message, and invokes the command interpreter.

```
%OPSRT-I-WBREAK, WBREAK encountered

OPS5>
```

Format

WBREAK *[element-pattern]*

Argument

element-pattern
An element pattern is similar to a working-memory element, and can include a class name, scalar attributes specified with values, and a vector attribute specified with a value. You can specify one or more element patterns. If you specify the name of an element pattern for which a breakpoint is not set, the command sets a breakpoint. If you specify the name of an element pattern for which a breakpoint is set, the command deletes the breakpoint.

The argument is optional. If you do not specify the name of a production, the command displays the names of the productions for which breakpoints are set.

Example

The following command displays the names of the working-memory elements for which breakpoints are set:

```
OPS5>WBREAK
CHECK ^DATE 14 NOV 1988
```

Suppose a breakpoint is not set for the element pattern CHECK ^DATE 14 NOV 1988. The following command will set a breakpoint for that element:

```
OPS5>WBREAK CHECK ^DATE 14 NOV 1988
```

To delete the breakpoint, use the same command:

```
OPS5>WBREAK CHECK ^DATE 14 NOV 1988
```

WM

Displays the working-memory elements whose time tags are specified.

When the command displays a working-memory element, the output includes the following information:

- Time tag
- Name of the production that created the element
- List of atoms in the element

The system displays this information in the following format:

time-tag [production-name] (class-name attribute-1 value-1 attribute-2 value-2...)

Format

WM *[time-tag...]*

Argument

time-tag
An integer that represents the time tag of a working-memory element the command is to display. You can specify one or more time tags.

The argument is optional. If you do not specify a time tag, the command displays all the elements in working memory.

Example

The following command displays all the elements in working memory:

```
OPS5>WM
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988)
2 [NIL] (CHECK ^NUMBER 103 ^AMOUNT 22.45 ^COUNTED NO ^DATE 14 NOV 1988)
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
6 [WHAT-DATE](COUNT ^VALUE 0)
7 [WHAT-DATE](REPLY ^DATE 14 NOV 1988)
```

The following command displays the working-memory elements whose time tags are 3 and 4:

```
OPS5>WM 3 4
3 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED NO ^DATE 14 NOV 1988)
4 [NIL] (CHECK ^NUMBER 108 ^AMOUNT 13.10 ^COUNTED NO ^DATE 25 NOV 1988)
```

Chapter 13

Support Routines

This chapter describes the VAX OPS5 support routines you can include in programs written in other VAX languages. These routines enable such programs to communicate with VAX OPS5 programs. To understand the descriptions in this chapter, you should be familiar with the procedure for calling external routines (described in the *VAX OPS5 User's Guide*). Table 13–1 lists the names of the support routines and gives brief descriptions. The rest of the chapter provides detailed descriptions and examples presented alphabetically by name.

Some of the routines store values in a buffer called the result element. This buffer is used to hold atoms that are arguments to actions, and atoms that will constitute a working-memory element, until the element is added to working memory.

The syntax used to describe the routines is:

routine-name (argument1, argument2...)

The parentheses are included even when no arguments are required. For example:

```
OPS$ASSERT ()
```

For information about how to use the support routines, see the *VAX OPS5 User's Guide*.

Table 13–1: Summary of Support Routines

Support Routine	Description
OPS$ACCEPT	Reads input from the terminal or a file and places the input in the result element
OPS$ACCEPTLINE	Reads a line of input from the terminal or a file and places the input in the result element
OPS$ASSERT	Adds the current result element to working memory, creating a new working-memory element
OPS$ATOM	Creates and returns a unique symbolic atom
OPS$CANCEL_RUN	Causes execution of recognize-act cycles to be suspended at the end of the next cycle
OPS$CLEAR	Clears the run-time system
OPS$COMPLETION	Synchronizes completion routines
OPS$CRLF	Places an end-of-line character string in the result element

(continued on next page)

Table 13–1 (Cont.): Summary of Support Routines

Support Routine	Description
OPS$CVAF	Converts a floating-point atom to a floating-point number and returns the result
OPS$CVAN	Converts an integer atom to an integer and returns the result
OPS$CVFA	Converts a floating-point number to a floating-point atom and returns the result
OPS$CVNA	Converts an integer to an integer atom and returns the result
OPS$EQL	Compares two atoms for equality and returns a Boolean result
OPS$FLOATING	Tests whether an atom is a floating-point atom and returns a Boolean result
OPS$GENATOM	Places a unique symbolic atom in the result element
OPS$HALT	Stops program execution
OPS$IFILE	Returns the address of an input file's record management service (RMS) record access block (RAB)
OPS$INITIALIZE	Initializes the VAX OPS5 program
OPS$INTEGER	Tests whether an atom is an integer atom and returns a Boolean result
OPS$INTERN	Translates a character string to a symbolic atom and returns the result
OPS$LITBIND	Returns the integer atom representing the field associated with an attribute name
OPS$LITVAL	Places the integer atom representing the field associated with an attribute name in the result element
OPS$OFILE	Returns the address of an output file's RMS record access block (RAB)
OPS$PARAMETER	Returns an argument value from the result element
OPS$PARAMETERCOUNT	Returns the integer representing the number of argument values stored in the result element
OPS$PNAME	Translates a symbolic atom to a character string and returns the string's length
OPS$RESET	Deletes all atoms currently stored in the result element
OPS$RUN	Causes execution of recognize-act cycles
OPS$STARTUP	Executes the VAX OPS5 program's STARTUP statement
OPS$SYMBOL	Tests whether an atom is a symbol and returns a Boolean result
OPS$TAB	Specifies the field in which the next entry to the result element is to be placed
OPS$VALUE	Places an atom in the result element
OPS$WARNING	Displays a warning message on the terminal
OPS$WRITE	Displays on the terminal the atoms currently in the result element, or writes the atoms to a file

OPS$ACCEPT

Reads input from the terminal or a file and places the input in the result element. To determine whether the input is an atom or a list of atoms (enclosed in parentheses), the routine checks the first printing character in the input. The routine assumes that any first character other than a parenthesis indicates an atom and places the atom in the result element; it assumes that an unquoted opening parenthesis indicates a list. It reads the atoms in the list until it encounters a closing parenthesis, removes the outermost pair of parentheses, and puts the atoms in the result element.

By default, the OPS$ACCEPT routine reads input from the terminal. If you want the routine to read input from a file, specify the routine with a file identifier, or set a file as the default source with the DEFAULT action or command (see Section 5.8.2) before calling the external routine.

When the OPS$ACCEPT routine reads past the end of a file, the routine places the symbol END–OF–FILE in the result element.

Using this routine is similar to using the ACCEPT function described in Chapter 11.

Format

OPS$ACCEPT *([file-id])*

Argument

file-id
The file identifier (symbolic atom) of the source file from which input is to be read. This argument is optional. If you do not specify a file identifier or if the name you specify is not associated with an open input file, the input is read from the current default for the ACCEPT function (set with the DEFAULT action or command).

Return Value

None.

OPS$ACCEPTLINE

Reads a line of input terminated by a carriage-return character, discarding all unquoted parentheses.

If some of the atoms on a line of input have already been read, the line of input is defined as all the remaining atoms on the current line, that is, from the current position to the next carriage-return (CRLF) character; if no atoms have been read from the current line, the input line is defined as the whole of the current line; if all the atoms on the current line have been read, the input line is the line following the current line.

If the input line contains no atoms, the default values are used.

By default, the OPS$ACCEPTLINE routine reads input from the terminal. If you want the routine to read input from a file, call the routine with the file identifier, or set a file as the default for source with the DEFAULT action or command (see Section 5.8.2) before calling the external routine.

Using this routine is similar to using the ACCEPTLINE function described in Chapter 11.

Format

OPS$ACCEPTLINE *([file-id], [default-value],...)*

Arguments

file-id
The file identifier (symbolic atom) of the source file from which input is to be read. This argument is optional. If you do not specify a file identifier, or if the identifier you specify is not associated with an open input file, the input is read from the current default for the ACCEPTLINE function (set with the DEFAULT action or command).

default-value
An atom the OPS$ACCEPTLINE routine is to place in the result element if the routine reads:

- A line of input from the terminal that consists of only a carriage return
- A file that consists of a line of only spaces and tabs
- Past the end of a file

A default value is optional and you can specify one or more of them.

Return Value

None.

OPS$ASSERT

Adds the current result element to working memory, creating a new working-memory element.

Format

OPS$ASSERT *()*

Return Value

None.

OPS$ATOM

Creates and returns a symbolic atom. The first occurrence of the routine generates the atom G:1. The second atom the routine generates is G:2, the third atom is G:3, and so on. For information about how to use system-generated atoms, see the *VAX OPS5 User's Guide*.

Using this routine is similar to using the GENATOM function described in Chapter 11.

Format

OPS$ATOM *()*

Return Value

A unique symbolic atom.

OPS$CANCEL_RUN

Stops program execution. The VAX OPS5 run-time system stops executing recognize-act cycles when the current cycle ends and, if VAX OPS5 informational messages are enabled, displays the following message and invokes the VAX OPS5 command interpreter when execution stops.

```
%OPSRT-I-CANCELED, OPS$CANCEL_RUN used
OPS5>
```

If HALT is enabled, the run-time system returns control to the operating system, or to the calling program if the VAX OPS5 program was called as a subroutine. To enable and disable HALT, use the ENABLE and DISABLE commands described in Chapter 12.

Format

OPS$CANCEL_RUN *()*

Return Values

Two values:

OPS$_NORMAL	The request to stop execution has been successfully queued.
OPS$_NORUN	The run-time system was not executing.

OPS$CLEAR

Clears the run-time system: it removes all working-memory elements from working memory, clears the conflict set, and resets the time-tag counter. It can be called at any time after OPS$INITIALIZE has been called, and it can be called repeatedly. However, it cannot be called from external routines while the VAX OPS5 system is running.

NOTE

This routine is normally called from a main program that calls a VAX OPS5 program as a subroutine.

Format

OPS$CLEAR *()*

Return Values

Three values:

OPS$_NORMAL	The run-time system has been successfully cleared.
OPS$_NOTINI	The OPS$INITIALIZE service has not yet been called.
OPS$_RUNNING	The VAX OPS5 system is currently running recognize-act cycles.

OPS$COMPLETION

Specifies or queries the address of a routine that is to be called at the end of the current recognize-act cycle. (This is normally used to change working memory following an interrupt. The VAX OPS5 program will not produce correct results if working memory is altered during the wrong phase of the recognize-act cycle.)

For more information about completion routines, see the *VAX OPS5 User's Guide*.

Format

OPS$COMPLETION *([address])*

Argument

address
The address (longword) of a completion routine, or 0. If you specify an address, the run-time system calls the completion routine at the end of the current recognize-act cycle. If you specify 0, the recognize-act cycle continues executing the VAX OPS5 program. The argument is optional.

Return Value

The value specified as an argument. If you specified the address of a completion routine, the support routine returns that address. If you specified 0, the support routine returns 0. If you do not specify an argument, the support routine returns the last argument value specified with the support routine.

OPS$CRLF

Places an end-of-line symbol (a carriage return followed by a line feed) in the result element. Use this routine before a call to the OPS$WRITE routine if you want output to be displayed on a new line.

Using this routine is similar to using the CRLF function described in Chapter 11.

Format

OPS$CRLF *()*

Return Value

None.

OPS$CVAF

Converts a floating-point atom to a floating-point number and returns the result.

Format

OPS$CVAF *(floating-point-atom)*

Argument

floating-point-atom
The floating-point atom to be converted to a floating-point number.

Return Value

A floating-point number of type VAX F_float data (see the *VAX Architecture Handbook*).

OPS$CVAN

Converts an integer atom to an integer and returns the result.

Format

OPS$CVAN *(integer-atom)*

Argument

integer-atom
The integer atom to be converted to an integer.

Return Value

An integer.

OPS$CVFA

Converts a floating-point number to a floating-point atom and returns the result.

Format

OPS$CVFA *(floating-point-number)*

Argument

floating-point-number
The floating-point number to be converted to a floating-point atom. The number you specify must be VAX F_float data (see the *VAX Architecture Handbook*).

Return Value

A floating-point atom.

OPS$CVNA

Converts an integer to an integer atom and returns the result.

Format

OPS$CVNA *(integer)*

Argument

integer
The integer to be converted to an integer atom.

Return Value

An integer atom.

OPS$EQL

Compares two atoms for equality and returns a Boolean result. The result is true (indicated by 1) if one of the following statements is true:

- The atoms are both symbols that consist of the same characters in the same order.
- The atoms are the same integer.
- The atoms are the same floating-point number.

The result is false (indicated by 0) in all other cases.

Format

OPS$EQL *(atom1, atom2)*

Arguments

atom1
The first atom to be compared.

atom2
The second atom to be compared.

Return Value

The integer 1 if the result is true and 0 if the result is false.

OPS$FLOATING

Tests whether an atom is a floating-point atom and returns a Boolean result. If the atom is a floating-point atom, the result is true (indicated by 1). Otherwise, the result is false (indicated by 0).

Format

OPS$FLOATING *(atom)*

Argument

atom
The atom to be tested.

Return Value

The integer 1 if the result is true and 0 if the result is false.

OPS$GENATOM

Places a symbolic atom in the result element. The routine calls the OPS$ATOM routine to create the new atom. An example of such an atom is G:1. After that routine has created the atom, the OPS$GENATOM routine calls the OPS$VALUE routine to place that atom in the result element.

For information about how to use system-generated atoms, see the *VAX OPS5 User's Guide.*

Using this routine is similar to using the GENATOM function described in Chapter 11.

Format

OPS$GENATOM *()*

Return Value

None.

OPS$HALT

Stops program execution. The VAX OPS5 run-time system stops executing recognize-act cycles when the current cycle ends and, if VAX OPS5 informational messages are enabled, displays the following message and invokes the VAX OPS5 command interpreter when execution stops:

```
%OPSRT-I-HALTED, HALT -- right-hand-side action
```

```
OPS5>
```

If HALT is enabled, the run-time system returns control to the operating system, or to the calling program if the VAX OPS5 program was called as a subroutine. To enable and disable HALT, use the ENABLE and DISABLE commands described in Chapter 12.

Format

OPS$HALT *()*

Return Value

None.

OPS$IFILE

Returns the address of the record access block (RAB) of an input file's record management services (RMS). The external routine can then use the RAB to read input from that file.

For information about RABs, see the *VMS Record Management Services Manual.*

Format

OPS$IFILE *(file-id)*

Argument

file-id
The file identifier (symbolic atom) for a file opened for input in the VAX OPS5 program.

Return Value

The RAB address of the file associated with the specified file identifier. If the specified name is not associated with an open input file, the support routine returns 0.

OPS$INITIALIZE

Initializes a VAX OPS5 program. It must be called before calls are made to any other support routine. (This should be used only if the VAX OPS5 program is called as a subroutine.)

Format

OPS$INITIALIZE *()*

Return Values

Two values:

OPS$_NORMAL	The run-time system has been successfully initialized.
OPS$_NOINI	The OPS$INITIALIZE routine has already been called.

OPS$INTEGER

Tests whether an atom is an integer atom and returns a Boolean result. If the atom is an integer atom, the result is true (indicated by 1). Otherwise, the result is false (indicated by 0).

Format

OPS$INTEGER *(atom)*

Argument

atom
The atom to be tested.

Return Value

The integer 1 if the result is true and 0 if the result is false.

OPS$INTERN

Translates a character string to a symbolic atom and returns the result.

Format

OPS$INTERN *(address, length)*

Arguments

address
The address of the character string to be translated.

length
The length of the character string to be translated.

Return Value

A symbolic atom.

OPS$LITBIND

Returns the integer atom representing the field associated with an attribute name.

Format

OPS$LITBIND *(attribute-name)*

Argument

attribute-name
A symbolic atom that represents the name of the attribute whose field is to be returned. The attribute name that you specify must have been declared in the VAX OPS5 program with a LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration.

Return Value

The integer atom representing the field associated with the specified attribute name. If the attribute name is not declared, the support routine returns that name.

OPS$LITVAL

Places the integer atom representing the field associated with an attribute name in the result element. The routine calls the OPS$LITBIND routine, which returns the integer representing the field, and then calls the OPS$VALUE routine to place that integer in the result element.

Using this function is similar to using the LITVAL function described in Chapter 11.

Format

OPS$LITVAL *(attribute-name)*

Argument

attribute-name
A symbolic atom that represents the name of the attribute whose field is to be placed in the result element. The attribute name that you specify must have been declared in the VAX OPS5 program with a LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration.

Return Value

None.

OPS$OFILE

Returns the address of the record access block (RAB) of an input file's record management services (RMS). The external routine can then use the RAB to write output to that file.

For information about RABs, see the *VMS Record Management Services Manual.*

Format

OPS$OFILE *(file-id)*

Argument

file-id
The file identifier (symbolic atom) for a file opened for output in the VAX OPS5 program.

Return Value

The RAB address of the file associated with the specified file identifier. If the specified name is not associated with an open output file, the support routine returns 0.

OPS$PARAMETER

Returns an argument value from the result element.

When a call to an external subroutine contains arguments, the VAX OPS5 run-time system places the argument values in the result element, starting in field 1. Use the OPS$PARAMETER routine to retrieve the argument values from the result element.

Format

OPS$PARAMETER *(field)*

Argument

field
An integer indicating the field from which an argument value is to be returned. The integer must be less than or equal to the integer returned by the OPS$PARAMETERCOUNT routine.

Return Value

The atom stored in the specified field.

OPS$PARAMETERCOUNT

Returns the integer representing the number of argument values stored in the result element.

When a call to an external subroutine contains arguments, the VAX OPS5 run-time system places the argument values in the result element, starting in field 1. Use the OPS$PARAMETERCOUNT routine to return an integer that indicates the number of argument values the run-time system placed in the result element for a particular call.

Format

OPS$PARAMETERCOUNT *()*

Return Value

The integer representing the number of arguments stored in the result element.

OPS$PNAME

Translates a symbolic atom to a character string and returns the string's length.

Format

OPS$PNAME *(symbolic-atom, address, length)*

Arguments

symbolic-atom
The symbolic atom to be translated to a character string.

address
The address of the buffer to which the character string is to be copied.

length
An integer that represents the maximum number of characters in the character string to be copied (the size of the buffer).

Return Value

The integer representing the number of characters in the character string (even if the routine does not copy the entire string). If you do not specify a symbolic atom for the first argument, the routine returns 0, and a character string is not copied. If the length you specify is less than the number of characters in the string being copied, the routine copies only the number of characters indicated by the value of the length argument.

OPS$RESET

Deletes all atoms currently stored in the result element, filling each field of the result element with NIL, and setting the TAB to the first field. (See the description of the OPS$TAB routine later in this chapter.)

Format

OPS$RESET *()*

Return Value

None.

OPS$RUN

Causes recognize-act cycles to be executed. It can be called at any time after OPS$INITIALIZE has been called, and it can be called repeatedly. However, it cannot be called from external routines if the VAX OPS5 system is currently running.

NOTE

This routine is normally called from a main program that calls a VAX OPS5 program as a subroutine.

Format

OPS$RUN *([count])*

Argument

count
This argument is optional. If you do not specify a value, the program will run until the conflict set is empty, a HALT action is executed, the user types Ctrl/C, or the OPS$CANCEL_RUN service is called.

If you specify a value that is less than zero, the program will run until the number of cycles specified in the STARTUP statement has been performed. If the number you specify is greater than or equal to zero, the program will run until that number of cycles has been performed.

Return Values

Seven values:

OPS$_CANCELED	The OPS$CANCEL_RUN routine was called.
OPS$_EMPTYCS	The conflict set is empty.
OPS$_EXIT	The user typed Ctrl/Z or the EXIT command at the OPS5> prompt.
OPS$_HALTED	A right-hand-side HALT action has been executed.
OPS$_NOTINI	The OPS$INITIALIZE service has not yet been called.
OPS$_PAUSE	The requested number of recognize-act cycles has been completed.
OPS$_RUNNING	The VAX OPS5 system is currently executing recognize-act cycles.

OPS$STARTUP

Executes the VAX OPS5 STARTUP statement. It can be called at any time after OPS$INITIALIZE has been called, and it can be called repeatedly.

Format

OPS$STARTUP *()*

Return Values

Three values:

OPS$_NORMAL	The STARTUP statement has been executed.
OPS$_NOSTARTUP	There is no STARTUP statement in the program.
OPS$5_NOTINI	The OPS$INITIALIZE service has not yet been called.

OPS$SYMBOL

Tests whether an atom is a symbol and returns a Boolean result. If the atom is a symbol, the result is true (indicated by 1). Otherwise, the result is false (indicated by 0).

Format

OPS$SYMBOL *(atom)*

Argument

atom
The atom to be tested.

Return Value

The integer 1 if the result is true and 0 if the result is false.

OPS$TAB

Specifies the field in which the next entry to the result element is to be placed.

Format

OPS$TAB *(field)*

Argument

field
An integer atom or attribute name (symbolic atom) that represents the field in which the next entry to the result element is to be placed. If you specify an integer atom, the routine specifies that the next entry be placed in a field that corresponds to that integer. If you specify an attribute name, the routine specifies that the next entry be placed in the field associated with that name.

Return Value

None.

OPS$VALUE

Places an atom in the result element. The field of the result element in which the atom is placed is determined as follows:

- If the call to the OPS$VALUE routine follows a call to the OPS$TAB routine, the OPS$VALUE routine places the atom in the field specified by the call to OPS$TAB.
- If the OPS$VALUE routine follows the OPS$RESET routine, the OPS$VALUE routine places the atom in the first field.
- Otherwise, the OPS$VALUE routine places the atom in the field following the last field in which an entry was made. Thus, consecutive calls to this routine create a vector.

Format

OPS$VALUE *(atom)*

Argument

atom
The atom to be placed in the result element.

Return Value

None.

OPS$WARNING

Displays a warning message on the terminal.

For more information about warning messages, see the *VAX OPS5 User's Guide*.

Format

OPS$WARNING *(address, length, [atom],...)*

Arguments

address
The address of the first byte of the character string in which the warning message is stored.

length
An integer that represents the length of the character string that stores the warning message.

atom
An atom the OPS$WARNING routine is to display in front of the warning message. This argument is optional. You can specify one or more atoms.

Return Value

None.

OPS$WRITE

Displays the atoms currently in the result element on the terminal or writes the atoms to a file. If the first field of the result element contains a file identifier, the routine writes the remaining atoms to the file associated with that name. Otherwise, the routine writes the atoms to the default destination, which is the terminal unless you change the default with the DEFAULT action or command (see Section 5.8.2).

Using this routine is similar to using the WRITE action described in Chapter 10.

Format

OPS$WRITE *()*

Return Value

None.

Appendix A

%INCLUDE Compiler Directive

VAX OPS5 provides one compiler directive, %INCLUDE, which allows you to include another VAX OPS5 program file in the current compilation.

A.1 Format

%INCLUDE *filespec*

A.2 Argument

filespec
The filespec you give must be a valid VMS file specification for the file that you want to include. If you do not specify a complete file specification, the compiler uses the default device and directory, and the file type OPS.

A.3 Example

```
%INCLUDE TESTPROG.OPS
```

NOTES

The VAX OPS5 compiler includes the file that you specify in the program compilation at the point where the %INCLUDE directive appears, and prints the included code in the program listing file, if the compilation produces one.

The file you specify for inclusion can itself contain a %INCLUDE directive.

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon, enclose the specification in vertical bars (| |).

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A. and Puerto Rico*	1–800–DIGITAL	

* Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

From	Call	Write
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order desk
Internal orders (for software documentation)	—	Software Distribution Center (SDC) Digital Equipment Corporation Westminster, MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Serv. (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro, MA 01532

Reader's Comments

VAX OPS5
Reference Manual
AA–EZ19C–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What I like best about this manual: ______________________________

____

What I like least about this manual: ______________________________

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

______ ______________________________

My additional comments or suggestions for improving this manual:

____

____

____

Please indicate the type of user/reader that you most nearly represent:

☐ Administrative Support	☐ Scientist/Engineer
☐ Computer Operator	☐ Software Support
☐ Educator/Trainer	☐ System Manager
☐ Programmer/Analyst	☐ Other (please specify) ______________
☐ Sales	

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

10/87

— — 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

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
PKO3-1/30D
129 PARKER STREET
MAYNARD, MA 01754-2198

— — Do Not Tear — Fold Here —

Cut Along Dotted Line