VAX OPS5 User's Guide

Order Number: AA–EZ18C–TE

May 1989

This document explains how to create, compile, execute, and debug VAX OPS5 programs.

If you want to use a VAXstation to perform those tasks, you should also read the *VAX OPS5 Development Environment User's Guide* and keep this document as a reference for the VAX OPS5 compiler and run-time system diagnostic messages.

Revision/Update Information:	This document supersedes the *VAX OPS5 User's Guide*, AA–EZ18B–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–S1162

Contents

Preface

Manual Objectives

VAX OPS5 is an extended implementation of the OPS5 language definition.

The *VAX OPS5 User's Guide* explains how to create, run, and revise VAX OPS5 programs on VMS operating systems. It also provides information about VAX OPS5 features that pertain to the VAX OPS5 compiler's and run-time system's interfaces with the programmer, user, and operating system.

If you want to use a VAXstation, you should also read the *VAX OPS5 Development Environment User's Guide* and keep this document as a reference for the VAX OPS5 compiler and run-time system diagnostic messages.

Intended Audience

This manual is for programmers who have a working knowledge of VAX OPS5. Knowledge of VMS and familiarity with the *Introduction to VMS* are recommended. Some sections of this manual require a more extensive understanding of the operating system. In such sections, you are directed to other manuals for additional information.

Structure of This Document

This manual consists of six chapters and an appendix.

Chapters 1 and 2 introduce VAX OPS5 architecture and program development on VMS.

Chapters 3, 4, and 5 explain how to compile, link, execute, and debug VAX OPS5 programs. Chapter 3 also explains how to create shareable VAX OPS5 programs.

Chapter 6 describes special programming techniques. Chapter 6 explains how to use the VAX OPS5 command interpreter to control input and output operations.

Appendix A summarizes the diagnostic messages generated by the VAX OPS5 compiler and run-time system.

Associated Documents

For tutorial information, see the self-paced instruction (SPI) course *The Artificial Intelligence Education Series: VAX OPS5*. For information about the VAX OPS5 programming language, see the *VAX OPS5 Reference Manual.*

The following documents provide additional information:

VAX OPS5 Version 3.0 Release Notes (on-line)
VAX OPS5 Development Environment User's Guide
VMS DECwindows User's Guide
Introduction to VMS
VMS DCL Dictionary
Guide to Using VMS Command Procedures
VMS Utilities Reference Volumes
VAX Architecture Handbook
Introduction to VMS System Routines
VAX 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.

Conventions

The following conventions are used in this document:

Convention	Meaning
[ ]	Square brackets usually enclose items that are optional. For example: `[/qualifier...]` However, when square brackets are used in the syntax of a directory name in a VMS file specification, the brackets must be included in the syntax.
...	A horizontal ellipsis means that the item preceding the ellipsis can be repeated. For example: `/qualifier...`
. . .	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.

Convention	Meaning
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.
Ctrl/X	Ctrl/X indicates a control key sequence where X can be any alphabetic character. Press the key labeled Ctrl while you simultaneously press another key. For example: Ctrl/C or Ctrl/Y The system echoes control key sequences as ^ followed by a character X. Therefore, in examples of output, Ctrl/X is shown as ^X. For example: Ctrl/C is displayed as ^C Ctrl/Y is displayed as ^Y

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 unless otherwise stated.

Chapter 1
VAX OPS5 Architecture

VAX OPS5 is used in the field of artificial intelligence, to develop expert systems, and in the field of cognitive psychology. VAX OPS5 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
- Facilities to aid knowledge representation
- Easy-to-learn syntax

This chapter describes the architecture of the VAX OPS5 system software, which consists of a compiler and a run-time system.

NOTE

For additional information, the following text is 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.

1.1 Compiler

The VAX OPS5 compiler converts VAX OPS5 module source files into object code for subsequent linking with modules written in VAX OPS5, or other VAX languages, and the VAX OPS5 run-time system. If you have a single VAX OPS5 source file to compile and execute, the compiler can generate a command file that controls linking, and submit the command file for execution.

For more information about the compiler and how to control compilation, see Chapter 3.

1.2 Run-Time System

The VAX OPS5 run-time system controls the execution of VAX OPS5 programs and consists of a recognize-act cycle, command interpreter, and compiler.

1.2.1 Recognize-Act Cycle

The recognize-act cycle consists of four steps:

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

While finding matches, the system compares working-memory elements with the condition elements on the left-hand side of each production. When working-memory elements match all the condition elements, the production is ready for execution. A production ready for execution is placed into a conflict set. There may be one or more productions in the conflict set at any time while a program is running. The system selects one of the ready productions, executes the actions on its right-hand side, and begins the cycle again. When there are no more matches, the conflict set is empty and the program halts.

For more information about the recognize-act cycle, see the *VAX OPS5 Reference Manual*.

1.2.2 Command Interpreter

The VAX OPS5 command interpreter lets you use commands to:

- Execute recognize-act cycles
- Debug VAX OPS5 programs
- Control input and output
- Perform file operations
- Call external routines
- Control loops

For more information on the command interpreter, see Section 4.2.

1.2.3 Run-Time System Compiler

In VAX OPS5, you can add productions to an executing program. The compiler is considered part of the run-time system when compiling these new productions because compilation occurs while the program is executing. For more information about adding productions to an executing program, see the description of the BUILD action in the *VAX OPS5 Reference Manual*.

Chapter 2

Introduction to Program Development on VMS

This chapter shows how the VMS command language (DCL) is used to:

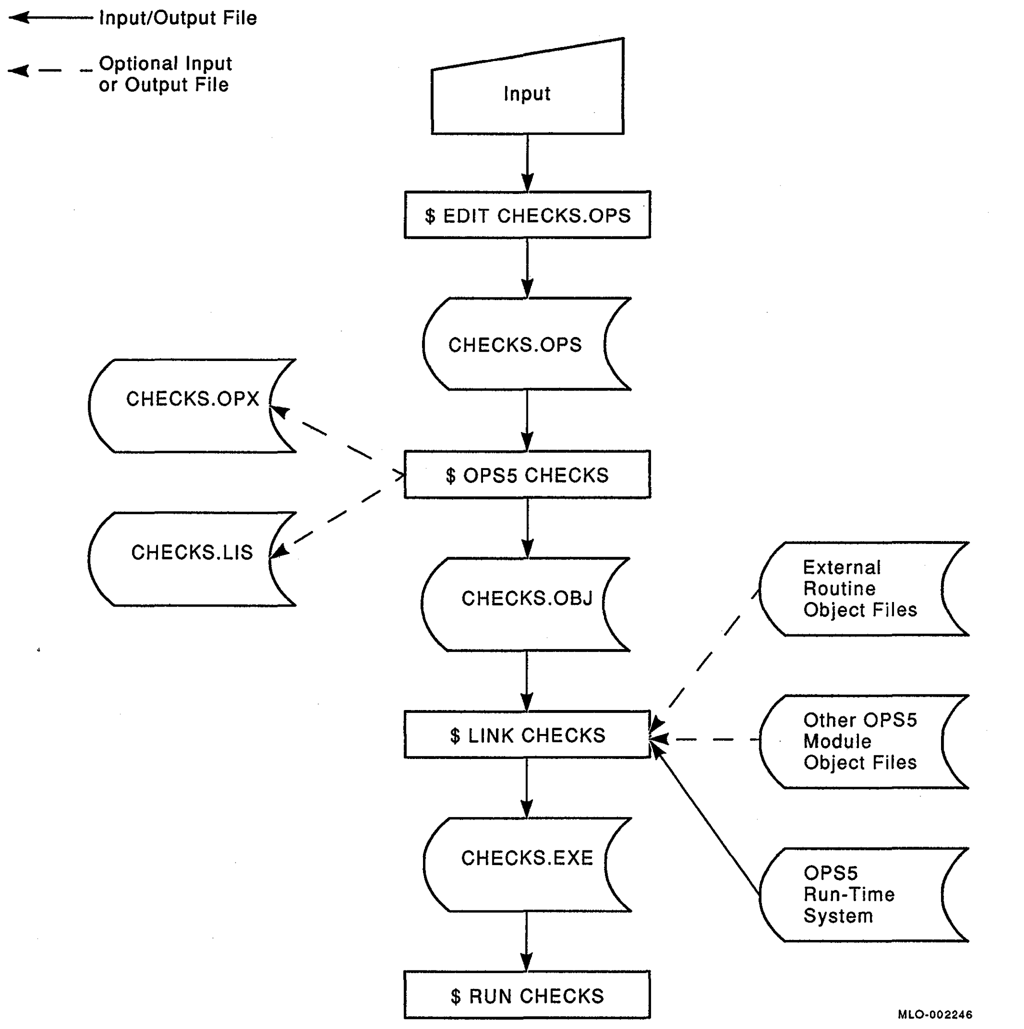
- Create, compile, link, and execute VAX OPS5 programs
- Specify input and output files for VAX OPS5 commands and programs

For an introduction to these concepts, see the *Introduction to VMS*. For detailed definitions of commands and file specifications, see the *VMS DCL Dictionary*.

2.1 Commands for Program Development

Figure 2–1 illustrates the DCL commands you use to create and execute the VAX OPS5 program CHECKS.OPS. The commands in the figure are shown in their simplest forms. You can also use qualifiers with the commands to request special processing or to indicate a special type of input file.

Figure 2–1: Commands for VAX OPS5 Program Development



1. The EDIT command invokes the system editor to create a file containing VAX OPS5 source code.

2. The VAX OPS5 command invokes the VAX OPS5 compiler to process the source code and verify that there are no syntax errors or violations of the language rules. If no errors occur, the compiler generates an object file and an optional listing file.

   If you compile a single source file to produce an executable image, the compiler generates an object file, a command file and an optional listing file. The command file invokes the VMS Linker, which links the object file with the VAX OPS5 run-time system to produce an executable image, and then deletes the object files and itself (unless otherwise specified). If errors occur, correct them and recompile. If only warnings occur, you can run the program, but the results might not be what you expect.

3. The LINK command invokes the VMS Linker, which links the object file with other object modules created with VAX OPS5 or other VAX languages, and with the VAX OPS5 run-time system to produce an executable program image.

If the LINK process fails or produces warning messages, you may have specified the command wrongly. Try again. If the problem persists, check the modules specified for the errors shown in the warning messages. Correct the source code, recompile, and link again.

4. The RUN command executes a program image.

   If your program fails or produces unexpected output, it probably contains an error. Correct the source code, recompile, link again, and reexecute.

2.2 Using the Help Facility

Enter the HELP command for on-line information about a command, its parameters, and its qualifiers. For example:

```
$ HELP OPS5
```

The help facility describes the OPS5 command, lists the additional information available, and prompts you for an OPS5 subtopic:

```
OPS5 Subtopic?
```

If you want the help facility to display information about a qualifier, respond to the prompt with the name of the qualifier. For example:

```
OPS5 Subtopic? /LIST
```

You can get the same information about the /LIST qualifier by typing the following command:

```
$ HELP OPS5/LIST
```

For further information on DCL and VAX OPS5 commands and qualifiers, see the *VMS DCL Concepts Manual*, *Guide to Using VMS Command Procedures*, and the *VMS DCL Dictionary*.

2.3 Specifying Files

To define a VAX OPS5 source file, give it a unique name and the file type OPS. Other components of the file specification can default to the system-supplied and command-supplied names. The following DCL commands show how to compile, link, and execute a VAX OPS5 program consisting of two source modules:

```
$ EDIT CHECKS.OPS
$ EDIT BALANCE.OPS
$ OPS5/CREATE/INDEX=CHECKS CHECKS
$ OPS5/INDEX=CHECKS BALANCE
$ LINK/EXECUTABLE=CHECKS.EXE CHECKS,BALANCE,OPSINTERP/LIBRARY
$ RUN CHECKS
```

The following example shows how to compile a single source file VAX OPS5 program into an executable image:

```
$ EDIT CHECKS.OPS
$ OPS5/EXECUTABLE=CHECKS CHECKS
$ RUN CHECKS
```

The default file types for the OPS5 command are:

- EXE—for an executable image
- LIS—for a listing file
- OBJ—for an object file

- OPS—for a source file
- OPX—for an index file

The LINK command assumes OBJ as the default file type.

The RUN command assumes EXE to be the executable image default file type.

Table 2–1 summarizes the default values for file specification components.

Table 2–1: File Specification System Defaults

Component	Default Value
Directory	User's current default directory
File name	Input: None
	Output: Same as input file; if no input file is specified, there is no default
Type	Input:
	OPS—Source file
	Input and Output:
	OPX—Modular compilation index file OBJ—Modular compilation object file
	Output:
	EXE—Executable or shareable image LIS—Listing file
Version	Input: Highest existing version number
	Output: If no existing version, 1; if existing version, highest version number plus 1

Chapter 3

Compiling and Linking VAX OPS5 Programs

This chapter explains how to use the OPS5 command to compile VAX OPS5 programs.

You compile a VAX OPS5 source file with the DCL command OPS5, which invokes the VAX OPS5 compiler to process the source code and verify that it contains no syntax errors or violations of the language rules. If no errors occur, the compiler generates an object file, an index file and, optionally, an index file and a listing file.

It is best to split large programs into a number of modules. You can write a module in VAX OPS5 or another VAX language. With the VAX OPS5 compiler, you can compile each VAX OPS5 module separately, then link the modules to each other and to the VAX OPS5 run-time system to produce an executable image.

If you have a single VAX OPS5 source file that you want to compile into an executable image, you can use the VAX OPS5 compiler's /EXECUTABLE switch to combine the compile and link steps.

3.1 VAX OPS5 Compiler Size Restrictions

The VAX OPS5 compiler enforces the following restrictions:

- An atom name can have a maximum of 256 characters.
- A working-memory element (WME) can have a maximum of 255 scalar attributes and one vector attribute.
- A declared vector attribute is assigned to field 256 in a WME.
- A vector can store a maximum of 127 atoms.
- A production can have a maximum of 32 positive condition elements and an unlimited number of negative condition elements.
- The right-hand side (RHS) of a production can have a maximum of 64 CBIND actions.
- A declaration, production, or STARTUP statement can have a maximum of 1024 actions, commands, or definitions.

3.2 The OPS5 Command

The syntax for the OPS5 command and its qualifiers is:

```
$ OPS5[/qualifier...] file-spec,...
```

qualifier
See Table 3–1 later in this chapter.

Additional qualifiers that can be used with the VAX OPS5 Development Environment are described in the *VAX OPS5 Development Environment User's Guide.*

file-spec
Names one or more source files to be compiled into either an object file (the default) or an executable image (for single source file programs only).

If you specify more than one source file, separate the file names with either a comma (,) or a plus sign (+). The compiler concatenates the files and compiles them as one source file. If you enter the OPS5 command without a file specification, the operating system prompts you for a file:

```
$ OPS5
_File:
```

The default file type for source files is OPS. You can specify another name for an object file, a listing file, or an executable image, by including a file specification with the /OBJECT, /LIST, or /EXECUTABLE qualifier. For example, the following command generates the listing file CHECKSLIST.LIS and the object file CHECKS.OBJ:

```
$ OPS5/LIST=CHECKSLIST CHECKS
```

The following example compiles the source file module CHECKS.OPS and generates the object file MYCHECKS.OBJ:

```
$ OPS5/OBJECT=MYCHECKS CHECKS
```

The following example compiles the single source file program CHECKS.OPS, generating the executable image MYCHECKS.EXE and the object file CHECKS.OBJ:

```
$ OPS5/EXECUTABLE=MYCHECKS CHECKS
```

3.3 Using the %INCLUDE Compiler Directive

Just as complex programs can be broken down into many parts, large source files can be split up into different text files. These separate text files can be included into one text file at compile time by using the %INCLUDE directive in a source file.

The %INCLUDE directive makes it easy to create a source file, perhaps of LITERALIZE declarations, to be used by different modules in a project. Each module can %INCLUDE the common declaration file. The compiler opens and reads the declaration file into the module in memory at the point where it finds the %INCLUDE directive, compiling the result as if it were one file.

The format of the %INCLUDE directive is:

```
%INCLUDE file-spec
```

where file-spec is any valid VMS file specification for the text file to be included.

These restrictions apply to the %INCLUDE directive:

- Following an %INCLUDE directive, you can only enter a comment on the same line. If you put anything else on the line, it will be ignored and you will get a warning from the compiler.
- The VAX OPS5 compiler includes the specified source file at the point where the %INCLUDE directive appears and prints the included source text at that point in the program listing file (if one is specified).
- Files accessed by %INCLUDE may themselves contain %INCLUDE directives.
- If you do not specify a complete file-spec, VAX OPS5 assumes the default device, default directory, and the file type OPS.
- If you want to use a file-spec that has a semicolon and version number, place the file-spec between vertical bars, as shown below:

```
%INCLUDE | CHECKS.DAT;2 |
```

3.4 Using VAX OPS5 Qualifiers to Control the Compiler and Linker

The default settings for the OPS5 command produce an object file. However, by specifying the command with qualifiers, you can precisely control the compilation of a program. For example, you can tell the compiler to generate a listing file by specifying the /LIST qualifier.

When you include a qualifier in a list of files to be concatenated, the qualifier affects all the files in the list.

Table 3–1 lists the ways you can control compilation, with the corresponding qualifiers (defaults are in bold print). The sections following the table explain how to use the qualifiers.

Table 3–1: VAX OPS5 Compiler Qualifiers

Qualifier	**Operation**
/CREATE	Create a new index file for a modular compilation. This option is only valid with the /INDEX_FILE qualifier.
/NOCREATE	Do not create an index file. Modular compilation is not required.
/ENTRY	Produce an image entry point for this module. The /ENTRY qualifier only affects the first module of any program.
/NOENTRY	Do not produce an image entry point for this module.
/EXECUTABLE[=file-spec]	Produce and name the executable image for a program consisting of only one source module.
/NOEXECUTABLE	Do not produce an executable image.
/INDEX_FILE=file-spec	Name the index file for a modular compilation. The default file type is OPX.
/NOINDEX_FILE	Do not generate an index file. Modular compilation is not required.
/LIST[=file-spec] (batch)	Generate a listing file. The default file type is LIS. This is the default in batch mode.

(continued on next page)

Table 3–1 (Cont.): VAX OPS5 Compiler Qualifiers

Qualifier	Operation
/NOLIST (interactive)	Do not generate a listing file. This is the default in interactive mode.
/MACHINE_CODE	Include machine code in the listing file. This option is only valid with the /LIST qualifier.
/NOMACHINE_CODE	Do not include machine code in the listing file.
/OBJECT[=file-spec]	Name the object file generated by the compiler.
/NOOBJECT	Do not generate an object file. The compiler only performs syntax checking.

3.4.1 Creating an Index File for Modular Compilation (/[NO]CREATE)

The /CREATE qualifier creates a new index file for a modular compilation. It must be used along with the /INDEX_FILE qualifier. The default file name for the index file produced is INDEXFILE. The default file type is OPX. The index file is created either in the directory to which you assign the logical name OPS$USERLIB or the current directory if OPS$USERLIB is not assigned.

This qualifier only affects the first module of a modular compilation. For further information, see the description of the /INDEX_FILE qualifier in Section 3.4.4.

The following example shows the command used to compile the module CHECKS.OPS as the first module of a modular compilation. This command creates the index file BALANCE.OPX:

```
$ OPS5/CREATE/INDEX_FILE=BALANCE CHECKS
```

3.4.2 Producing a Program Entry Point (/[NO]ENTRY)

This qualifier affects the entry point for the VAX OPS5 program to be compiled. It is only valid for the first module in a modular compilation. If you are compiling any other module, or using /EXECUTABLE, this qualifier has no effect.

A program's entry point is the point in the program's executable image where it starts executing. By default, the VAX OPS5 compiler places the entry point for the program in the first module you compile.

If you write a program in another language and want to call a subroutine written in VAX OPS5, use the VAX OPS5 compiler's /NOENTRY qualifier when compiling the VAX OPS5 module. This prevents the definition of a program entry point in the VAX OPS5 module.

Subject to a few restrictions, described in Section 3.7, you can choose any module as the first module of a modular compilation.

The following example compiles CHECKS.OPS into an object file, prevents the assignment of a program entry point to it (allowing it to be used as a subroutine in another program), and does not create an index file.

```
$ OPS5/NOENTRY CHECKS
```

This qualifier has no effect on the compilation of the second or subsequent modules in a multi-module VAX OPS5 program.

3.4.3 Generating and Naming an Executable Image (/[NO]EXECUTABLE)

If your program consists of only one VAX OPS5 source file, you can generate and name an executable image by using the /EXECUTABLE qualifier.

If you do not supply a file specification, the executable image produced by the linker defaults to the name of the first VAX OPS5 source file specified in the command, your default directory, and an EXE file type.

You can use the /EXECUTABLE qualifier to specify a specific device, directory, or file name for the executable image. For example, suppose you want to compile a source file named CHECKS.OPS but you want the name of the executable image to be MYCHECKS.EXE rather than CHECKS.EXE. Specify the /EXECUTABLE qualifier with the output file name MYCHECKS:

```
$ OPS5/EXECUTABLE=MYCHECKS CHECKS
```

3.4.4 Defining an Index File (/[NO]INDEX_FILE)

Use the /INDEX_FILE qualifier as follows:

- Alone, with a file name, to tell the compiler which modular compilation a module belongs to
- With the /CREATE qualifier, to tell the compiler that the module specified is the first module of a modular compilation

The following example compiles the module CHECKS.OPS as a module of the modular compilation defined by the index file BALANCE.OPX:

```
$ OPS5/INDEX_FILE=BALANCE CHECKS
```

The next example compiles the module CHECKS.OPS as the first module of the modular compilation defined in the index file CHECKSLIST.OPX and creates the index file.

```
$ OPS5/CREATE/INDEX_FILE=CHECKSLIST CHECKS
```

For further information on /INDEX_FILE, see Section 3.7.

The compiler assumes /NOINDEX_FILE as the default.

3.4.5 Producing a Listing File (/[NO]LIST)

A listing file is useful for debugging because it provides information about errors the compiler detects during compilation. In interactive mode, the compiler produces a listing file only if you specify the /LIST qualifier. The compiler will not produce a listing file by default. The default file specification consists of the name of the first VAX OPS5 source file specified in the command, your default directory, and a LIS file type. For example, the following command causes the compiler to produce the listing file CHECKS.LIS:

```
$ OPS5/LIST CHECKS
```

If you want to give the listing file a different name, use the /LIST qualifier with a file specification. For example, to compile the program CHECKS.OPS, naming the listing file CHECKSLIST.LIS, use the following command:

```
$ OPS5/LIST=CHECKSLIST CHECKS
```

In batch mode, the compiler produces a listing file by default. To suppress the listing file, use the /NOLIST qualifier.

3.4.6 Including Machine Code in the Listing File (/[NO]MACHINE_CODE)

By default, the listing file consists of source code and a compilation summary. However, the listing file also includes machine code if you specify the /MACHINE_CODE qualifier with the /LIST qualifier. For example:

```
$ OPS5/LIST/MACHINE_CODE CHECKS
```

3.4.7 Naming Object Files (/[NO]OBJECT[=file-spec])

To produce an object file with a different name from your source file, use the /OBJECT[=file-spec] switch, where file-spec is the new name for the object file. The following command compiles the source file BALANCE.OPS into the object file CHECKS.OBJ:

```
$ OPS5 BALANCE/OBJECT=CHECKS
```

To stop generation of an object file use the /NOOBJECT switch. The following command compiles the source file CHECKS.OPS and generates a listing, but no object file:

```
$ OPS5/LIST/NOOBJECT CHECKS
```

3.5 Linking VAX OPS5 Programs

When you have successfully compiled all the modules of your program, you will have to link them together and to the VAX OPS5 run-time system to produce an executable image.

Two forms of the run-time system are provided with the VAX OPS5 compiler kit:

- OPSINTERP.OLB—An object library
- OPSINTERP.EXE—A shareable image

If you wish to create a program that can run on systems without the VAX OPS5 run-time system installed, link your program modules to OPSINTERP.OLB. For example:

```
$ LINK MYPROG,CHECKS,BALANCE,OPSINTERP/LIBRARY
```

On the other hand, if your program will only run on systems that have the run-time system installed, use the LINK command's OPTIONS section to link the program modules to OPSINTERP.EXE. For example:

```
$ LINK MYPROG,CHECKS,BALANCE,OPSINTERP/OPTIONS
```

When VAX OPS5 is installed, the logical OPSINTERP is defined as OPS$LIBRARY:OPSINTERP.

3.6 Compiling, Linking, Executing, and Debugging External Routines

The procedure for compiling an external routine depends on the programming language in which the routine is written. See the appropriate language user's guide for instructions on how to compile an external routine. The compilation produces an object file whose name you can include in the command you use to link the VAX OPS5 program.

After you have compiled the VAX OPS5 and external routines, link the VAX OPS5 object files with the object files of the external routines to produce an executable image. For example, where the external routine object file is STOCKSUB.OBJ, compile the VAX OPS5 modules to object code:

```
$ OPS5/CREATE/INDEX_FILE=STOCKCTRL STOCKINIT
$ OPS5/INDEX_FILE=STOCKCTRL DOSTOCK
```

Then LINK the VAX OPS5 and STOCKSUB object modules together and to the appropriate run-time system.

```
$ LINK STOCKINIT,DOSTOCK,STOCKSUB,OPSINTERP/OPTIONS
```

The VAX OPS5 compiler compiles the source files STOCKINIT.OPS and DOSTOCK.OPS. The linker then links the object files generated during compilation with the object file STOCKSUB.OBJ created by another compiler.

If the compiler, or linker, generates errors, correct the errors and recompile, or relink, the program. Execute the program when it compiles and links without error. If execution errors occur, debug, recompile, relink and execute the program again. For information about compiling, linking, executing, and debugging VAX OPS5 programs, see Chapters 3, 4, and 5.

Use the CALL command at the command interpreter level and the VMS Debugger to debug external routines. For information about using the debugger, see Chapter 5 and the *VMS Debugger Manual*.

3.7 Performing a Modular Compilation

Many programs are too large and complex to be contained in one source file. These programs are usually broken down into logical sections, each of which performs a specific task, or group of tasks, within the program.

You can store these sections in multiple source files or modules, which can be compiled separately. Since the amount of code being compiled is reduced, the program can be compiled more efficiently. After you have compiled and debugged all the modules, you can link them to create an executable image.

Modular compilation also makes the debugging process faster because finding errors in small pieces of code is easier than finding errors in a large program.

There are a few simple rules to follow when performing a modular compilation:

- Declare all attributes, class names and external routines before use, using LITERAL, LITERALIZE, and EXTERNAL statements. The compiler makes all declarations made in a module available to all the modules compiled after it.
- Compile the first module with the /CREATE and /INDEX_FILE qualifiers.
- Compile the subsequent modules with the /INDEX_FILE qualifier, specifying the same index file name as with the first module.

When you compile the first module of a program, the compiler creates an index file. The compilation of each subsequent module adds information to the index file. The index file stores the following information:

- The number of each module
- Declared attribute names
- Declared external-routine names
- Catcher names
- Whether a module contains a STARTUP statement
- Production names

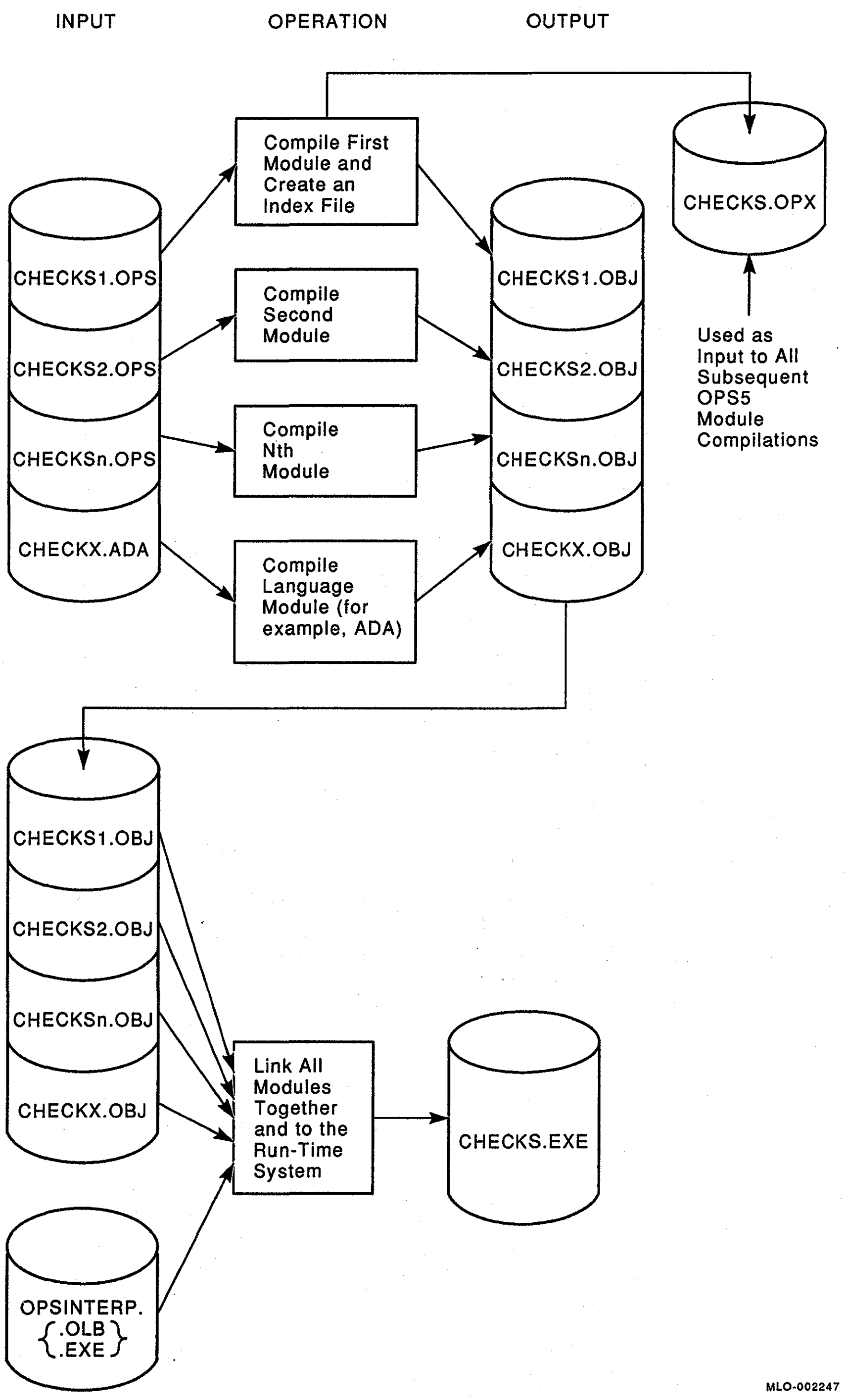
The index file allows the compiler to check for duplication (for example, names or declarations) and allows modules to use declarations made in previously compiled modules. Figure 3–1 illustrates the steps needed to perform a modular compilation.

Use the following command sequence to perform a modular compilation:

```
$ OPS5 module_1/INDEX_FILE=filename/CREATE
$ OPS5 module_2/INDEX_FILE=filename
              .
              .
              .
$ OPS5 module_n/INDEX_FILE=filename
$ LINK MODULE1,...MODULEn,OPSINTERP/OPTIONS
```

You can specify additional qualifiers in either the OPS5 command (as described in Section 3.4) or the LINK command (see the *VMS Linker Utility Manual*).

Figure 3–1: Modular Compilation



If the program consists of a large number of modules, it is often useful to save the object files in a library. The following example shows how:

```
$ LIBRARY/CREATE MODLIB.OLB
$ OPS5/INDEX_FILE=CHECKS/CREATE CHECKS1
$ LIBRARY MODLIB.OLB CHECKS1.OBJ
$ OPS5/INDEX_FILE=CHECKS CHECKS2
$ LIBRARY MODLIB.OLB CHECKS2.OBJ
                    .
                    .
                    .
$ OPS5/INDEX_FILE=CHECKS CHECKSn
$ LIBRARY MODLIB.OLB CHECKSn.OBJ
$ LINK MODLIB.OLB/INCLUDE=(CHECKSn.OBJ), MODLIB/LIB, OPSINTERP/OPTIONS
```

The /INCLUDE qualifier names the most recently compiled module. This module references all other modules in the library, causing the linker to include them in the executable image file.

1. The directory defined as OPS$USERLIB is the directory in which the compiler places the index file for the modular compilation. The compiler will use the current directory if OPS$USERLIB is not defined.
2. When you compile the first module, specify the /INDEX_FILE and /CREATE qualifiers. The /INDEX_FILE qualifier specifies the file name for the index file. The compiler assigns the file type OPX to the index file. In Figure 3–1, the command specifies the file name CHECKS.

   The /CREATE qualifier causes the compiler to create the index file in OPS$USERLIB even if a file of that name already exists. The compiler then places the first module's object file in the default directory or the directory you specified.
3. When you compile the remaining modules, specify the /INDEX_FILE qualifier. The /INDEX_FILE qualifier specifies in which index file to place the compilation information. Specify the same index file name when you compile each module. To update the contents of an index file for a module, you must specify the name of the file with the /INDEX_FILE qualifier. In Figure 3–1, the file name CHECKS is specified. Therefore, when you execute the commands, the compiler references CHECKS.OPX when compiling each module. Each time you compile a new module, index information for that module is added to the index file.

   If a module contains errors, you can edit and recompile the module. If you change a declaration in a module, you must recompile that module and all the modules originally compiled after it in the program. It is usually safest to recompile the entire program if a declaration changes.
4. When you have successfully compiled all the modules, use the VMS LINK command to link the VAX OPS5 object files (and external object files, if any) together into an executable image.

   To compile a single source file into an executable image, you should use the /EXECUTABLE qualifier. For example:

   ```
   $ OPS5/EXECUTABLE=MYCHECKS CHECKS
   ```

   This command compiles the source file CHECKS.OPS into the executable image MYCHECKS.EXE. This can only be done with a program contained in one source file and thus is not practical for large or complex programs.

3.8 Creating Shareable VAX OPS5 Programs

A shareable image produced by the VMS Linker cannot be directly executed by the DCL command RUN. A shareable image serves as input to another linking operation that produces an executable image.

Shareable images are used for:

- Sharing a single physical copy of a set of procedures and data among more than one application program
- Linking very large applications by breaking down a program into smaller segments
- Modifying one section of a large program without having to relink the entire program

3.8.1 Creating a Shareable Image

To create a shareable image, follow the instructions given in the *VMS Linker Utility Manual.*

3.8.2 Installing a Shareable Image

You can use the VAX OPS5 language to create large expert systems. If such a system is frequently used, creating and installing a shareable image for the system reduces the overall system requirements for main physical memory.

After you have created a shareable image, install the image by following the procedures outlined in the *VMS Install Utility Manual.*

3.8.3 Calling a Shareable Image

After a shareable image has been installed, you can call the image as a subroutine from programs written in other VAX programming languages.

You call a shareable image by specifying its entry point with the call routine appropriate to the language. For example, suppose you create a shareable image CHECKSHR.EXE having the entry point CHECKS. The following MACRO routine CALLCHECKS.MAR calls the shareable image:

```
.TITLE    CALLCHECKS
.EXTERNAL CHECKS
.PSECT    MYPSECT,EXE,NOWRT
.ENTRY    DOIT,^M<>
CALLS     #0,G^CHECKS               ; Note general-mode
                                    ; addressing
RET
.END      DOIT
```

To assemble this MACRO routine, follow the procedures given in the *VMS DCL Dictionary* under the MACRO command. You can also refer to the *VAX MACRO and Instruction Set Reference Manual.*

Link the object module of the assembled MACRO routine with the VAX OPS5 shareable image as follows:

```
$ LINK CALLCHECKS,SYS$INPUT:/OPTIONS
SYS$LIBRARY:CHECKSHR.EXE/SHAREABLE
Ctrl/Z
```

Enter a carriage return after the /OPTIONS qualifier and again after the /SHAREABLE qualifier. Then, enter Ctrl/Z to indicate the End-of-File.

This command produces the executable image CALLCHECKS.EXE, which calls the VAX OPS5 shareable image CHECKSHR.EXE. (The procedure for linking shareable images is explained in the *VMS Linker Utility Manual.*)

If you do not place a copy of the shareable image in the system library, inform the image activator where to find the image before you execute the routine that calls the image. The image activator is a set of VMS system procedures that prepares an image for execution. To inform the activator where the image is, define a logical name that includes the device, directory, and file name of the image. For example, if the shareable image CHECKSHR.EXE resides on the device DBA1 in directory [SMITH.CHECKS], define a logical name as follows:

```
$ DEFINE CHECKSHR DBA1:[SMITH.CHECKS]CHECKSHR
```

Now you can execute the image CALLCHECKS.EXE. For information on calling external programs and routines from a VAX OPS5 program, see the *VAX OPS5 Reference Manual.*

Chapter 4

Executing VAX OPS5 Programs

VAX OPS5 programs are executed under the control of the VAX OPS5 run-time system, which processes the executable image by executing recognize-act cycles. This chapter explains how to run an executable image and how to control program execution by submitting commands interactively to the run-time system's command interpreter. This chapter also explains how to interrupt recognize-act cycles to invoke the command interpreter.

4.1 Running Programs

There are two ways to run a VAX OPS5 program:

- Use the DCL command RUN on the executable image. If the file is found, the operating system passes control to the VAX OPS5 run-time system, which either starts executing recognize-act cycles or invokes the command interpreter. If the program contains a startup statement that includes the VAX OPS5 command RUN, the system immediately starts executing recognize-act cycles. If there is no startup statement, the system invokes the command interpreter.
- Use the DCL command RUN to load and execute a program that calls the VAX OPS5 program. When the VAX OPS5 program is called, control passes to the VAX OPS5 run-time system, under the same conditions as above, and returns to the calling program when the VAX OPS5 routine terminates—provided the command DISABLE HALT is in the VAX OPS5 STARTUP statement. If the VAX OPS5 routine does not contain a DISABLE HALT statement, the routine will return control to the VAX OPS5 run-time system, but not to the calling program.

  See the *VAX OPS5 Reference Manual* for more information on calling VAX OPS5 programs from other languages, and calling routines written in other languages from VAX OPS5 programs.

4.2 Using the VAX OPS5 Command Interpreter

The VAX OPS5 command interpreter lets you interactively control the execution of a program by entering VAX OPS5 commands. Use the VAX OPS5 commands to:

- Set up initial conditions
- Execute recognize-act cycles
- Restart VAX OPS5 programs
- Add statements, productions, and catchers to executable images

- Debug VAX OPS5 programs
- Control input and output
- Call external routines
- Control loops

Section 4.2.3 explains how you can use commands to set up initial conditions; Section 4.3 explains how to execute recognize-act cycles; Section 4.5 explains how to restart VAX OPS5 programs; and Section 4.6 explains how to add statements, productions, and catchers to executable images. Chapters 5 and 6 provide information on how to debug VAX OPS5 programs and control input and output. See the *VAX OPS5 Reference Manual* for information about controlling loops and calling external routines.

4.2.1 Entering VAX OPS5 Commands

When the run-time system is not executing a recognize-act cycle, and the run-time system's HALT switch is enabled, the system invokes the command interpreter and displays the prompt:

```
OPS5>
```

To enter a VAX OPS5 command, type the command, with arguments if appropriate, and then press the Return key. To extend a command over more than one line, you can either enclose it in parentheses (which you can optionally use for single-line commands also) or use the continuation character (–).

If you press the Return key without entering a command, the command interpreter redisplays the prompt.

An example of a single-line command entered, without parentheses, at the run-time system prompt is:

```
OPS5>CS
```

If you end a line with the continuation character, the command interpreter prompts you for the rest of the command each time you press the Return key. For example:

```
OPS5>MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO -
_OPS5>^DATE 2 NOV 1988
```

If you begin a command with a parenthesis, you must make sure you have the same number of left and right parentheses, as the interpreter will try to match each left parenthesis with a corresponding right parenthesis before accepting the command. An example of a command entered with parentheses is:

```
OPS5>(MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO -
_OPS5>^DATE 2 NOV 1988)
```

If you want to include an extra, unmatched, parenthesis in such a command, you must enclose it in quote characters, for example, | (|, otherwise the interpreter will count it as a parenthesis to be matched before it will allow you to end the command.

To specify arguments, which can be required or optional, enter the arguments after the command they modify. For example:

```
OPS5>MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988
```

The resulting working-memory element looks like this:

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

The command interpreter does not evaluate arguments. Therefore, command arguments cannot be variables or function calls.

If you do not specify a required argument, the run-time system displays a warning message and redisplays the command interpreter prompt. For example:

```
OPS5>MAKE
?OPSRT-W-NOARGS, No arguments specified
OPS5>
```

4.2.2 Exiting the Command Interpreter

Use the EXIT command or type Ctrl/Z to exit the command interpreter and return control to the operating system or the program calling the VAX OPS5 program.

```
OPS5>EXIT
$
```

The STARTUP statement in a VAX OPS5 program allows you to decide what happens when your program ends, or if it fails unexpectedly.

The DISABLE HALT command causes the command interpreter to exit, dropping you back to the DCL prompt level, or back into the program calling the VAX OPS5 routine.

The ENABLE HALT command brings you back to the OPS5 prompt when your program finishes executing recognize-act cycles.

```
OPS5>
```

4.2.3 Setting Up Initial Conditions

You can use VAX OPS5 commands to set up initial conditions, which include whether the run-time system should display messages, which conflict-resolution strategy the run-time system is to use, and the initial contents of working memory. You can set up such conditions by including the commands in a STARTUP statement or by specifying them in response to the command interpreter prompt.

4.2.3.1 Disabling and Enabling Run-Time Messages

By default, the run-time system prompts for commands and displays informational, warning, and fatal error messages. You can suppress these messages by using the DISABLE or ENABLE commands with the keywords WARNING or HALT respectively.

Specifying DISABLE HALT causes control to be returned to DCL, or a calling program, when the run-time system stops executing recognize-act cycles. For example:

```
OPS5>DISABLE HALT
OPS5>RUN
$
```

You can enable warning messages again by using the ENABLE command with the keyword WARNING. For example:

```
OPS5>ENABLE WARNING
```

The VAX OPS5 run-time system (or command interpreter) has ENABLE HALT set as the default. This means that the VAX OPS5 interpreter command prompt appears when recognize-act cycles stop executing.

For more information about run-time messages, see Appendix A.

4.2.3.2 Choosing a Conflict-Resolution Strategy

You can choose which of two conflict-resolution strategies you want the run-time system to use by using the STRATEGY command with the keyword LEX or MEA. For example, to choose the MEA strategy, type:

```
OPS5>STRATEGY MEA
```

To see which strategy is in use, use the STRATEGY command without a keyword.

See the *VAX OPS5 Reference Manual* for information about conflict resolution and the conflict-resolution strategies.

4.2.3.3 Initializing Working Memory

The run-time system cannot start executing recognize-act cycles until working memory contains an element. To create a working-memory element, use the MAKE command as follows:

```
OPS5>MAKE START
```

For more information about creating working-memory elements, see Section 5.1.3.1.

4.2.4 Using VAX OPS5 Command Files

You can store a list of VAX OPS5 commands in a file and execute it later by using the VAX OPS5 @ command. For example, the following command opens the file CHECKS.DAT and causes the command interpreter to execute the commands stored in that file:

```
OPS5>@ CHECKS.DAT
```

If the file you specify with the @ command contains information other than VAX OPS5 commands, the run-time system displays the following message:

```
%OPSRT-W-ILLCMD, Invalid command: AAAAAA
```

4.3 Executing Recognize-Act Cycles

The run-time system starts executing recognize-act cycles when it executes the VAX OPS5 command RUN. You can include the RUN command in a STARTUP statement, or type the command in response to the command interpreter prompt.

If you included the RUN command in a program's STARTUP statement, the run-time system starts executing recognize-act cycles when you enter the DCL command RUN. Suppose the VAX OPS5 program named CHECKS.OPS contains the following STARTUP statement:

```
(STARTUP
     (DISABLE HALT)
     (STRATEGY MEA)
     (MAKE START)
     (RUN))
```

The system starts executing recognize-act cycles when you type the following DCL command:

```
$ RUN CHECKS
```

If a program does not contain a STARTUP statement that includes the RUN command, you can enter the command in response to the command interpreter prompt as follows:

```
OPS5>RUN
```

You can control the number of recognize-act cycles the run-time system executes by specifying the RUN command with an integer. For example, to execute four recognize-act cycles, specify:

```
OPS5>(RUN 4)
```

4.4 Interrupting Recognize-Act Cycles

You can invoke the command interpreter while a program is executing by:

- Typing Ctrl/C
- Setting a breakpoint for a particular production in the executing program
- Including the HALT action in the right-hand side of a production in the program

4.4.1 Ctrl/C

If you type Ctrl/C while the run-time system is executing a program, the system completes the execution of the current recognize-act cycle, displays a message, and invokes the command interpreter. For example:

```
$ RUN CHECK

What date do you want to search for? 14 NOV 1988
Ctrl/C

%OPSRT-I-CTRLCNOTED, CTRL/C -- return to command interpreter

OPS5>
```

NOTE

If you type Ctrl/C during a recognize-act cycle that requires user input, you must supply the input before the interruption can occur.

4.4.2 Breakpoints

When the run-time system encounters a breakpoint, the system stops executing recognize-act cycles before executing the production for which the breakpoint is set, displays a message, and invokes the command interpreter.

```
%OPSRT-I-PBREAK, PBREAK encountered
OPS5>
```

You can set a breakpoint to make the system stop at the following points:

- Before a particular production in a program by using the PBREAK command

- When a particular working-memory element is made by using the WBREAK command

Refer to Section 5.1.1 for further information on the PBREAK and WBREAK commands.

4.4.3 HALT Actions

When the run-time system executes a HALT action, the system completes the current recognize-act cycle, displays the following message, and invokes the command interpreter.

```
%OPSRT-I-HALTED, HALT -- right-hand-side action
OPS5>
```

4.5 Restarting Programs

While debugging a VAX OPS5 program, frequently it is necessary to rerun the program from the beginning. Instead of exiting to the DCL level and then rerunning the program, you can use the RESTART command.

For example, the following command restarts the program CHECKS.EXE.

```
OPS5>RESTART CHECKS.EXE
```

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

4.6 Adding Statements, Productions, and Catchers to Executable Images

The BUILD command lets you add a STARTUP statement, a production, or a catcher to a running program that has been paused.

Use the following procedure.

1. Pause the running program.
2. Enter BUILD at the OPS5> prompt. OPS5 should display the _BUILD> prompt.
3. Enter the information you want to add to the program. Use as many lines as necessary.
4. Enter ENDBUILD or type Ctrl/Z. OPS5 should display the OPS5> prompt.
5. Enter RESTART to resume execution with the new information in effect.

For example, TEST.EXE is a program that does not yet have any constructs in it. 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 following message.

```
Dear Subscriber:
```

Chapter 5

Debugging VAX OPS5 Programs

You can use the VAX OPS5 debugging commands to find errors detected by the compiler and run-time system, and programming errors that you detect yourself. If a program calls external routines, you can use the VMS Debugger to find errors in the routines.

This chapter explains how to use VAX OPS5 commands to debug VAX OPS5 programs and briefly describes the VMS Debugger.

5.1 Using VAX OPS5 Debugging Commands

The command interpreter provides commands that perform debugging operations. Table 5–1 lists these commands with their corresponding operations. Tables 5–2 and 5–3 (later in this section) list the commands for setting trace levels within the VAX OPS5 command interpreter.

Table 5–1: Debugging Commands

Commands	Operation
ADDSTATE RESTORESTATE SAVESTATE	Save and restore the state of working memory and the conflict set
AFTER	Set the recognize-act counter for a catcher
BACK	Back up over recognize-act cycles
BUILD	Add a statement, production or catcher to a running program that has been paused.
CS NEXT	Display the contents of the conflict set
DISABLE ENABLE	Change the state of program operation
EXCISE	Disable productions
MAKE MODIFY REMOVE	Modify working memory
MATCHES	Display match information
PBREAK WBREAK	Use breakpoints

(continued on next page)

Table 5–1 (Cont.): Debugging Commands

Commands	Operation
PPWM WM	Display working-memory elements
REPORT	Use the Performance Measurement and Evaluation package
RESTART	Rerun a VAX OPS5 program without returning to DCL
RUN	Run a VAX OPS5 program or routine
STRATEGY	Choose which strategy to use
WATCH	Display trace information

The following sections explain how to use the commands. Detailed descriptions of the commands are provided in the *VAX OPS5 Reference Manual.*

5.1.1 Using Breakpoints

You can set, delete, and list breakpoints for productions by using the PBREAK command.

You can set, delete, and list breakpoints for working-memory elements by using the WBREAK command.

When the run-time system encounters a breakpoint, the system finishes executing the current recognize-act cycle, displays one of the following messages, and invokes the command interpreter to allow you to enter other debugging commands:

```
%OPSRT-I-PBREAK, PBREAK encountered
OPS5>
```

or

```
%OPSRT-I-WBREAK, WBREAK encountered
OPS5>
```

If a breakpoint is set for a production, the run-time system encounters the breakpoint before executing that production.

5.1.1.1 Setting and Deleting Breakpoints

To set or delete a breakpoint, specify the PBREAK or WBREAK command with the name of the production or WME. If you specify the name of a production or WME that does not have a breakpoint set, the command sets a breakpoint. If you specify the name of a production or WME that already has a breakpoint set, the command deletes the breakpoint.

For example, suppose the production FIND–CHECKS does not have a breakpoint set and the production COUNTED–CHECKS has a breakpoint set. The following command sets a breakpoint for the production FIND–CHECKS and deletes the breakpoint from the production COUNTED–CHECKS:

```
OPS5>PBREAK FIND-CHECKS COUNTED-CHECKS
```

Similarly, for the working-memory elements CHECK and DATE, CHECK has no breakpoint set and DATE has a breakpoint set. The following command sets a breakpoint for CHECK and deletes the breakpoint for DATE:

```
OPS5>WBREAK CHECK DATE
```

5.1.1.2 Listing Breakpoints

To see which productions have breakpoints set, use the PBREAK command without an argument. The following example shows that breakpoints are set for the productions WHAT–DATE and COUNTED–CHECKS:

```
OPS5>PBREAK
WHAT-DATE
COUNTED-CHECKS
```

To see which working-memory elements have breakpoints set, use the WBREAK command without an argument. The following example shows that breakpoints are set for the working-memory elements DATE and CHECK:

```
OPS5>WBREAK
DATE
CHECK
```

5.1.2 Displaying Working-Memory Elements

During a debugging session, you might want to examine the contents of working memory to make sure it contains correct information. Missing or erroneous working-memory elements can cause a production to be executed at the wrong time.

The WM and PPWM commands display the contents of working memory. The run-time system provides the following information about each working-memory element:

- The element's time tag
- The name of the production that added the element to working memory
- The element's class name, attributes, and the attributes' values

The run-time system displays this information in the following format:

```
time-tag [production-name] (class-name attribute-1 value-1
                                       attribute-2 value-2...)
```

When you use the MAKE command to create a working-memory element, the atom NIL appears in the second field of the display rather than the name of a production. For example:

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

5.1.2.1 Displaying the Contents of Working Memory

To display the entire contents of working memory, use either the WM or PPWM command without an argument. For example:

```
OPS5>WM
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 02 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)
```

5.1.2.2 Displaying Specific Working-Memory Elements

To display particular working-memory elements, specify the time tags of those elements with the WM command. The following example displays the working-memory elements that have time tags 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)
```

You can find out which time tags are assigned to working-memory elements by setting the run-time system's trace level to 2 or 3. For further information, see Section 5.1.6.

5.1.2.3 Displaying the Working-Memory Elements of a Class

To display the working-memory elements of a particular class, use the PPWM command with the name of that class. For example:

```
OPS5>PPWM CHECK
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 02 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)
```

5.1.2.4 Displaying Working-Memory Elements that Match Element Patterns

To display the working-memory elements that match a specific element pattern, use the PPWM command followed by the pattern you want to match. An element pattern can be a complete, or partial, specification of a working-memory element.

The following example displays the elements in working memory that match the partial element pattern ^DATE 14 NOV 1988.

```
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)
```

5.1.3 Modifying Working Memory

If working memory contains incorrect information, the productions in a program might not be executed as you anticipate. You can modify working memory by:

- Creating elements
- Deleting elements
- Changing the atoms in existing elements

5.1.3.1 Creating Working-Memory Elements

You can create new working-memory elements by using the MAKE command with a class name. You can optionally specify attributes with values. The following command creates an element with a class name CHECK:

```
OPS5>MAKE CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 2 NOV 1988
```

The element is stored in working memory as follows:

```
OPS5>WM
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 02 NOV 1988)
```

NIL is placed in the field that stores the name of the production that created the element.

5.1.3.2 Deleting Elements from Working Memory

The REMOVE command deletes elements from working memory. To delete specific elements, specify their time tags. The following example deletes the working-memory elements whose time tags are 3 and 4:

```
OPS5>REMOVE 3 4
```

To delete all working-memory elements, specify the command with an asterisk (*). For example:

```
OPS5>REMOVE *
```

5.1.3.3 Changing the Atoms in Working-Memory Elements

To change one or more atoms in a working-memory element, use the MODIFY command with the time tag of the element whose atoms you want to change and the attributes and new atoms.

Suppose, for example, working memory contains the following elements:

```
OPS5>WM
1 [NIL] (CHECK ^NUMBER 102 ^AMOUNT 10.06 ^COUNTED NO ^DATE 02 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 the atom for the attribute ^COUNTED of the working-memory element whose time tag is 3:

```
OPS5>MODIFY 3 ^COUNTED YES
```

The run-time system then rebuilds the element using the new atom and places the element in working memory with a new time tag:

```
9 [NIL] (CHECK ^NUMBER 104 ^AMOUNT 56.00 ^COUNTED YES ^DATE 14 NOV 1988)
```

5.1.4 Displaying Conflict Set Information

The conflict set is a list of matched working-memory elements and condition elements. You can display the contents of the conflict set or the instantiation of the next production to be executed.

Each member of the conflict set is an instantiation. An instantiation includes a production name and a list of time tags of the working-memory elements that satisfy the production's left-hand side. The format the run-time system uses to display an instantiation is:

```
production-name time-tag-1 time-tag-2...
```

For example:

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

The instantiations in the conflict set indicate which productions can be executed.

5.1.4.1 Displaying the Contents of the Conflict Set

The CS command displays the contents of the conflict set. For example:

```
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
```

5.1.4.2 Displaying the Instantiation of the Next Production To Be Executed

To display the instantiation of the next production the run-time system will execute, use the NEXT command. For example:

```
OPS5>NEXT
FIND-CHECKS 12 6 11
```

This shows that the next production the run-time system will execute is FIND–CHECKS. The production will be executed with its positive condition elements matched by the working-memory elements whose time tags are 12, 6, and 11.

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

If you are debugging a VAX OPS5 program and you need to stop execution to do something else, you might want to save the state of working memory and the conflict set as it exists at that time. By using VAX OPS5 commands you can save the state to a file and then restore it later.

5.1.5.1 Using SAVESTATE

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

```
OPS5>SAVESTATE CHECKS.DAT
```

5.1.5.2 Using ADDSTATE

Once you save the state of working memory and the conflict set in a file, you can add the contents of that file to the current state of working memory and the conflict set by using the ADDSTATE command. Suppose you used the SAVESTATE command to copy the state of working memory and the conflict set to the file CHECKS.DAT. You could use the following command to add the contents of that file to working memory and the conflict set:

```
OPS5>ADDSTATE CHECKS.DAT
```

The ADDSTATE action:

- Adds the working-memory elements in the saved state to current working memory
- Adds the instantiations arising from the working-memory elements in the saved state to the current conflict set

5.1.5.3 Using RESTORESTATE

The RESTORESTATE command clears and restores working memory and the conflict set to the state recorded in a file produced by the SAVESTATE command. Suppose you used the SAVESTATE command to copy the state of working memory and the conflict set to the file CHECKS.DAT. The following command restores working memory and the conflict set to the state recorded in the file CHECKS.DAT:

```
OPS5>RESTORESTATE CHECKS.DAT
```

The RESTORESTATE action:

- Clears current working memory
- Clears the current conflict set
- Restores working memory from the saved state
- Restores the conflict set from the saved state

NOTE

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

5.1.6 Displaying Trace Information

The VAX OPS5 run-time system displays trace information while executing a program. Trace information can be enabled for rules, working memory (WM), the conflict set (CS), and production memory (PM).

The amount of trace information the system displays depends on the current trace level. The integers 0 to 4 represent the five trace levels (see Tables 5–2 and 5–3); trace level 0 is the default.

Table 5–2: Trace Levels

Level	Effect
0	Disables all tracing
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

Table 5–3: Trace Keywords

Level	Effect
ALL	Enables rule, CS, PM, and WM tracing
CS	Enables tracing of instantiations into and out of the conflict set
PM	Enables tracing of productions into and out of production memory

(continued on next page)

Table 5–3 (Cont.): Trace Keywords

Level	Effect
RULE	Enables tracing of firing rules
WM	Enables tracing of working memory elements into and out of working memory
NOALL	Disables all tracing
NOCS	Disables conflict set tracing
NOPM	Disables production memory tracing
NORULE	Disables rule tracing
NOWM	Disables working memory tracing

5.1.6.1 Setting the Trace Level

To set the trace level, specify the WATCH command with an integer in the range 0 to 4, or with a trace keyword (or a list of keywords). For example, if you want the system to display as much trace information as possible, set the trace level to 4 or ALL.

```
OPS5>WATCH 4
```

or

```
OPS5>WATCH ALL
```

5.1.6.2 Displaying the Current Trace Level

To display the current trace level, use the WATCH command without an argument. The following example shows the current trace level is 2:

```
OPS5>WATCH
RULE  WM
```

5.1.6.3 Trace Level 1—RULE

At trace level 1, the run-time system displays a line of text, which includes the following information, each time the system executes a production:

- The number of productions executed
- The name of the production executed
- The time tags of the working-memory elements that matched the production's condition elements

The run-time system displays this information in the following format:

```
number:  production-name time-tag-1 time-tag-2...
```

In the following example, trace level 1 output is shown in **bold** print:

```
OPS5>RUN
1:  WHAT-DATE 10
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>>>2 NOV 1988
2:  FIND-CHECKS 12 8 11
```

```
Found check number 101 for $ 40.30 dated 2 NOV 1988
3:  FIND-CHECKS 12 1 15
                .
                .
                .
```

5.1.6.4 Trace Level 2—RULE and WM

When the trace level is 2, the run-time system displays the same information as it displays for trace level 1 plus the elements added to and deleted from working memory. The system displays the following information for each element:

- The element's time tag
- The name of the production that added or deleted the element
- The element's attributes and the attribute values

The system displays this information in the following format:

```
time-tag [production-name] (class-name
                             attribute-1 value-1 attribute-2 value-2...)
```

If an element was added to working memory, the time tag is preceded by the symbol =>WM:. If an element was deleted from working memory, the time tag is preceded by the symbol <=WM:.

In the following example, trace level 2 output is shown in **bold** print:

```
OPS5>WATCH 2
OPS5>RUN
1:  WHAT-DATE 10
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>>>
=>WM:  11 [WHAT-DATE] (COUNT ^VALUE 0) 2 NOV 1988
=>WM:  12 [WHAT-DATE] (REPLY ^DATE 2 NOV 1988)
<=WM:  10 [NIL] (START)
2:  FIND-CHECKS 12 8 11
                .
                .
                .
```

When all WMEs are deleted, for example, in a RESTORESTATE action (or in a REMOVE * action), the system displays:

```
<=WM:  ** All WMEs Removed **
```

5.1.6.5 Trace Level 3—RULE, WM and CS

When the trace level is 3, the run-time system displays the same information as trace level 2 plus the instantiations added to and deleted from the conflict set. The system displays the following information for an instantiation:

- The name of the production for which the instantiation was added or deleted
- The time tags of the working-memory elements that match condition elements in the production's left-hand side

The system displays this information in the following format:

```
production-name time-tag-1 time-tag-2...
```

If an instantiation was added to the conflict set, the production name is preceded by the symbol =>CS:. If an instantiation was deleted from the conflict set, the production name is preceded by the symbol <=CS:.

In the following example, trace level 3 output is shown in **bold** print:

```
OPS5>WATCH 3
OPS5>RUN
1:  WHAT-DATE 10
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>>>
=>WM:  11 [WHAT-DATE] (COUNT ^VALUE 0)2 NOV 1988
=>WM:  12 [WHAT-DATE] (REPLY ^DATE 2 NOV 1988)
=>CS:  FIND-CHECKS 12 1 11
=>CS:  FIND-CHECKS 12 8 11
<=WM:  10 [NIL] (START)
2:  FIND-CHECKS 12 8 11
                   .
                   .
                   .
```

When all conflict set elements are deleted, the system displays:

```
<=CS:  ** All CS Entries Removed **
```

5.1.6.6 Trace Level 4—RULE, WM, CS and PM

When the trace level is 4, the run-time system displays the same information as trace level 3 plus the productions added to and deleted from production memory. The system displays the following information for a production:

- The name of the production added to (using BUILD) or deleted from (using EXCISE) production memory.

The system displays this information in the following format:

```
symbol production-name
```

If a production was added to production memory, the production name is preceded by the symbol =>PM:. If a production was deleted from production memory, the production name is preceded by the symbol <=PM:.

In the following example, trace level 4 output is shown in **bold** print:

```
OPS5>WATCH 4
OPS5>EXCISE FIND-CHECKS
<=PM:  FIND-CHECKS
```

5.1.7 Displaying Match Information

You can display match information for specific productions by using the MATCHES command. By examining match information, you can detect whether condition elements are being matched correctly by working-memory elements.

Match information includes the time tags of working-memory elements that match condition elements in the productions you specify. First, the command displays the name of the production. Then the command lists the time tags for the working-memory elements that match the first condition element, the second condition element, and so on. The format of the match output follows:

```
>>> production-name <<<

*** matches for n ***
time-tag
    .
    .
    .
```

The n in this format 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.

More than one condition element can match the same working-memory element. Therefore, the MATCHES command also displays the time tags of the working-memory elements that match more than one condition element. The following format displays the time tags of working-memory elements that match two condition elements simultaneously:

```
>>> production-name <<<

*** matches for n m  ***
time-tag
    .
    .
    .
```

The n and m in this output represent the two condition elements that match the same working-memory elements.

The format for a production that contains three condition elements might look like the following:

```
>>> production-name <<<

*** matches for 1 ***
time-tag
    .
    .
    .
*** matches for 2 ***
time-tag
    .
    .
    .
*** matches for 1 2 ***
time-tag time-tag
    .
    .
    .
*** matches for 3 ***
time-tag
    .
    .
    .
*** matches for 1 2 3 ***
time-tag time-tag time-tag
    .
    .
    .
```

Suppose working memory contains the following elements:

```
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)
5 [WHAT-DATE] (COUNT ^VALUE 0)
6 [WHAT-DATE] (REPLY ^DATE 14 NOV 1988)
```

Look at this production:

```
(P FIND-CHECKS
     (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 1 DATE INF))
     (MODIFY <CHECK> ^COUNTED YES)
     (MODIFY <COUNTER> ^VALUE (COMPUTE 1 + <VALUE>)))
```

The following command displays the matches for this production:

```
OPS5>(MATCHES FIND-CHECKS)
>>> FIND-CHECKS <<<
*** matches for 1 ***
6
*** matches for 2 ***
1
2
3
4
*** matches for 1 2 ***
6 2
6 3
*** matches for 3 ***
5
*** matches for 1 2 3 ***
6 2 5
6 3 5
```

The output shows that the working-memory element with time tag 6 matches the first condition element and the working-memory elements with time tags 1, 2, 3, and 4 match the second condition element. The working-memory element with time tag 6 matches the first condition element while the working-memory elements with time tags 2 and 3 match the second condition element. The element with time tag 5 matches the third condition element. The working-memory element combinations with time tags of 6, 3, and 5, and 6, 2, and 5 together, both satisfy the first, second, and third condition elements.

5.1.8 Backing Up over Recognize-Act Cycles

You can restore and inspect a previous state of working memory and the conflict set by using the BACK command to back up over a specified number of recognize-act cycles. Specify the command with an integer, which indicates the number of cycles the system is to back up.

By default, the BACK command is disabled. You must enable it, using the ENABLE command with the keyword BACK, before you use it. For example:

```
OPS5>(ENABLE BACK)
```

If you specify the following command, the run-time system executes five recognize-act cycles and then displays the command interpreter prompt:

```
OPS5>(RUN 5)
1:  WHAT-DATE 10
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>>>14 NOV 1988
2:  FIND-CHECKS 12 6 11

Found check number 107 for $ 16.15 dated 14 NOV 1988
3:  FIND-CHECKS 12 5 15

Found check number 106 for $ 250.00 dated 14 NOV 1988
4:  FIND-CHECKS 12 4 19

Found check number 105 for $ 27.25 dated 14 NOV 1988
5:  FIND-CHECKS 12 3 23

Found check number 104 for $ 56.00 dated 14 NOV 1988
%OPSRT-I-PAUSED, Pause

OPS5>
```

The run-time system backs up two cycles if you specify:

```
OPS5>(BACK 2)
```

If you use the RUN command to execute another recognize-act cycle, you can see how the run-time system backed up two cycles.

```
OPS5>(RUN 1)
4:  FIND-CHECKS 12 4 19

Found check number 105 for $ 27.25 dated 14 NOV 1988
```

The maximum number of cycles that can be backed up is 64.

NOTE

The BACK command restores only working-memory elements that are created, deleted, or modified. The command does not reverse operations such as input and output operations and does not call external routines.

5.1.9 Disabling Productions

When debugging a VAX OPS5 program, you might want to disable productions that appear to be causing errors. To disable productions, use the EXCISE command. The command disables a production by deleting all instantiations for that production from the conflict set. The following example disables the productions FIND–CHECKS and COUNTED–CHECKS:

```
OPS5>(EXCISE FIND-CHECKS COUNTED-CHECKS)
```

NOTE

After you have disabled a production, the run-time system cannot place an instantiation of that production back into the conflict set.

5.1.10 Using the Performance Measurement and Evaluation Package

VAX OPS5 provides a Performance Measurement and Evaluation (PME) package, which you can use to collect data to monitor a program's execution. The package uses the data to create two reports: a timing CPU report and a cause report. The timing CPU report shows where a program is spending most of its time and includes the following information for each production:

- The number of times the production was executed
- The amount of CPU time (in 10-millisecond ticks) used to execute the left-hand side of the production

- The amount of CPU time (in 10-millisecond ticks) used to execute the right-hand side of the production

The timing CPU report shows where a program is spending most of its time.

The cause report lists the name of each production executed and the name of the production that caused it to be executed. A production causes another production to be executed if it creates working-memory elements that satisfy the other production's left-hand side. For example, if the production WHAT–DATE created working-memory elements that satisfied the left-hand side of the production FIND–CHECKS, the cause report lists the production names as follows:

```
PRODUCTION NAME             EFFECTING PRODUCTION NAME

FIND-CHECKS                 WHAT-DATE
```

The information the cause report provides lets you trace back through a program's execution. For example, while you are debugging a VAX OPS5 program, you might find that a particular production is being executed when it should not be. You can find out which production is creating working-memory elements that satisfy that production's left-hand side by referring to the cause report.

To enable the PME package, use the ENABLE command with the keyword TIMING. For example:

```
OPS5>(ENABLE TIMING)
```

Use the DISABLE command with the keyword TIMING to stop collecting performance data.

To create the reports, use the REPORT command with the keyword TIMING.

```
OPS5>(REPORT TIMING)
```

The timing CPU report is placed in the file TIMINGCPU.TXT and the cause report is placed in the file TIMINGCAU.TXT.

The following example shows the format of the reports:

```
Timing CPU report on 4-Apr-1988 15:08:14.00

PRODUCTION NAME         # FIRINGS      LHS TIME       RHS TIME

WHAT-DATE               3              8              24
FIND-CHECKS             15             34             72
COUNTED-CHECKS          3              10             18
STOP-COUNT              1              5              11

Cause report on 4-Apr-1988  15:08:14.00

PRODUCTION NAME         EFFECTING PRODUCTION NAME

FIND-CHECKS             WHAT-DATE
COUNTED-CHECKS          FIND-CHECKS
BUS-STOP                COUNTED-CHECKS
     .
     .
     .
```

5.2 VMS Debugger

If a VAX OPS5 program calls routines written in other VMS programming languages, you can use the VMS Debugger to debug the routines. The debugger lets you control the execution of the routine to monitor locations, change the contents of locations, check the sequence of program control, and locate and correct errors as they occur.

The VMS Debugger is interactive. The debugger also understands a variety of languages, such as VAX C, VAX FORTRAN and VAX PL/I, letting you change from one language to another during a debugging session.

With a module of external routines called BALANCE.PAS, you can invoke the VMS Debugger by compiling the VAX Pascal module with the DEBUG qualifier selected as follows:

```
$ PASCAL/DEBUG BALANCE
$ OPS5 CHECKS
```

You then link and run the modules in one of two ways. One way is:

```
$ LINK/EXE=MONEY/DEBUG CHECKS,BALANCE,OPSINTERP/OPTIONS
$ RUN MONEY
DBG>
```

The other way is:

```
$ LINK/EXE=MONEY CHECKS,BALANCE,OPSINTERP/OPTIONS
$ RUN/DEBUG MONEY
DBG>
```

For information about how to use the VMS Debugger, see the *VMS Debugger Reference Manual*.

Chapter 6

Controlling Input and Output

You can use the VAX OPS5 command interpreter to control where a VAX OPS5 program reads input and sends output. This chapter explains how to use the command interpreter to:

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

Detailed descriptions of the commands mentioned in this chapter are provided in the *VAX OPS5 Reference Manual.*

6.1 Opening Files

To open a file for input or output, use the OPENFILE command. Specify the command with a file identifier, a file specification, and the keyword IN, OUT, or APPEND, which indicates whether you are opening the file for reading or writing. The command opens the file and associates the file identifier with the file. For example, the following command opens the file CHECKS.DAT for reading and associates the file identifier CHECKSI with that file:

```
OPS5>OPENFILE CHECKSI CHECKS.DAT IN
```

To send output to the file CHECKS.LOG, open the file with the following command:

```
OPS5>OPENFILE CHECKSO CHECKS.LOG OUT
```

To open CHECKS.LOG for output and add data to the end of the file, open the file with the following command:

```
OPS5>OPENFILE CHECKSO CHECKS.LOG APPEND
```

NOTE

The comment character for VAX OPS5 is a semicolon (;). Therefore, if the VMS file specification you give includes a semicolon (;), enclose the specification within a pair of vertical bars (| |). For example:

```
OPS5>OPENFILE CHECKSO | CHECKS.LOG;5 | IN
```

Once a file is open 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)

The *VAX OPS5 Reference Manual* explains how to use the ACCEPT and ACCEPTLINE functions and the WRITE action to perform input and output operations.

6.2 Setting the Default Input Source and Output Destination

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

By default, input comes from SYS$INPUT. To set the source to a file, specify the DEFAULT command 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 example sets that input file to be the default source for input:

```
OPS5>DEFAULT CHECKSI ACCEPT
```

Once the default for input is set to a file, all input required by the ACCEPT and ACCEPTLINE functions is taken from that file, unless the ACCEPT command specifies an alternative source of input. To set the default back to the original value, specify the symbol NIL with the keyword ACCEPT.

```
OPS5>DEFAULT NIL ACCEPT
```

The terminal is also the default destination for output. To set the destination to a file, specify the DEFAULT command 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 Section 5.1.6) and the keyword WRITE sets the destination for the WRITE action. Suppose CHECKSO is the file identifier for an open output file. The following example sets that output file to be the default destination for trace output:

```
OPS5>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.

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

```
OPS5>DEFAULT NIL TRACE
```

6.3 Closing Files

To close files, specify the CLOSEFILE command with the file identifiers of the files you want to close. This command dissociates the file identifiers from the files. For example, to close files whose file identifiers are CHECKSI and CHECKSO, specify the following:

```
OPS5>(CLOSEFILE CHECKSI CHECKSO)
```

You no longer can use those identifiers with other actions, functions, or commands to perform input and output operations. To use the files again, you must reopen them.

Appendix A
Diagnostic Messages

This appendix lists and explains all the diagnostic messages produced by the VAX OPS5 compiler and run-time system. Each message appears here in the same form as on the display terminal or hard-copy terminal listing.

Some messages describe error conditions that you cannot resolve. When you receive a message of this type, you should submit a Software Performance Report (SPR) to Digital. An SPR is a form that customers who have warranties or who have support services can use to report faults in the software and suggest product improvements.

The following sections explain the order in which the messages are listed, describe the message formats, and explain how to control message display.

A.1 Message Order

The messages are arranged in alphabetical order. This appendix uses two conventions to alphabetize messages that contain special characters and general symbols that refer to parts of programs and commands.

The first convention deals with messages that contain general symbols, which appear in various messages and stand for specific names or values that are copied directly from programs and commands. They are not used for alphabetizing. These symbols are:

Symbol	Description
AAAAAA	A symbol the VAX OPS5 compiler or run-time system copies into messages from the program you are compiling or executing
BBBBBB	Same as above
DEV:FILNAM.TYP	A file specification the DCL command line interpreter copies into messages from a command
dd-mmm-yyyy	A date (day, month, year) the VAX OPS5 compiler includes in messages
hh:mm:ss	A time (hour, minutes, seconds) the VAX OPS5 compiler includes in messages
n	An integer the VAX OPS5 compiler or run-time system includes in messages
s.ss	A unit of time (seconds) the VAX OPS5 compiler includes in messages

The second convention is that the first digit or letter in the message is used for alphabetizing. General symbols and special characters are ignored. For example, the following message is alphabetized under the letter o:

```
%OPSCOMP-E-NOOPS5, No OPS5 files specified
```

A.2 Message Format

VAX OPS5 diagnostic messages appear in the following format:

```
%facility-l-ident, text
```

The message text is prefixed with the following:

- The name of the facility that is reporting the error. This can be OPSCOMP (compiler), OPSRT (run-time system) or LICENSE (License Management Utility). A percent sign (%) prefixes the facility.
- A severity level code (l) that indicates the severity of the message (see Table A–1)
- The message identification, which is usually an abbreviation of the message text
- Message text that briefly describes the condition that caused the message to be displayed

Table A–1: Message Severity Levels

Code	Level	Effect
I	Information	If the message is displayed by the compiler, execution continues. The condition might affect execution at a later time and might require future action. If the message is displayed by the run-time system, the message means program execution stopped.
W	Warning	Execution continues. A condition exists that might cause errors in execution. Corrective action might be necessary.
E	Error	Execution might terminate. An error exists that might cause other errors during execution. Corrective action is necessary.
F	Fatal/Severe	Execution terminates. A serious error exists. You must enter another command to continue processing.

When the compiler generates the following syntax error message, the listing file includes another message that identifies the compiler's recovery action.

```
%OPSCOMP-E-SYNTAX_ERROR, One of the following symbols was
expected: AAAAAA AAAAAA ...
```

The format of the syntax recovery messages follows:

```
;*** text ***
```

A.3 Controlling Message Display

By default, the operating system displays four message fields: facility, severity, identification, and text. You can control which fields of a message the system is to display by specifying the DCL command SET MESSAGE with the appropriate qualifiers. For example, if you want to include only message text, specify the following command:

```
$ SET MESSAGE/NOFACILITY/NOSEVERITY/NOIDENTIFICATION
```

A message display now looks like the following:

```
No OPS5 files specified
```

To reset the message display to include the facility, severity, and identification fields, specify the SET MESSAGE command with the positive forms of the qualifiers.

```
$ SET MESSAGE/FACILITY/SEVERITY/IDENTIFICATION
```

For more information about the SET MESSAGE command, see the *VMS DCL Dictionary*.

You can also control whether the run-time system displays messages by using the VAX OPS5 commands DISABLE and ENABLE. Specifying these commands with the keyword HALT controls run-time information messages. Specifying the commands with the keyword WARNING controls run-time messages produced for warnings and fatal errors. See Chapter 4 for more information about disabling and enabling VAX OPS5 run-time system messages.

A.4 List of VAX OPS5 Messages

;* "AAAAAA" inserted before symbol *****

Explanation: The compiler recovered from a syntax error by inserting the symbol AAAAAA before an erroneous symbol. This message appears only in a listing file and is associated with a previous syntax error message.

Your Response: The message is informational.

%OPSCOMP–E–BADBUILD, Attempted to BUILD AAAAAA construct

Explanation: Your program specified a BUILD action for one of the following declarations: EXTERNAL, LITERAL, LITERALIZE, VECTOR-ATTRIBUTE.

Your Response: Edit your program to remove or correct this invalid action and try again.

%OPSCOMP–E–BUFFOFLOW, Buffer overflow—source line too long. Line truncated

Explanation: A single line of source code exceeded 256 bytes, which is the maximum amount allowed for the compiler's input buffer. An overflow occurred.

Your Response: Break the line into smaller multiple lines, each less than 256 bytes.

%OPSCOMP-E-EXMEM, Exceeded memory allocation for BUILD action

Explanation: An attempt was made to build a rule, catcher, or startup statement so complex that it exceeded the 64K byte capacity of the compiler's internal object code buffer.

Your Response: Edit the program to simplify the production causing this error.

%OPSCOMP-E-INC_NONAME, No file name given. %INCLUDE ignored

Explanation: You used %INCLUDE without specifying a file.

Your Response: %INCLUDE must be followed by a file name.

%OPSCOMP-E-LOADINDEX, Unable to load index file

Explanation: 1. The /INDEXFILE qualifier was specified in the command without the /CREATE qualifier, and the specified index file does not exist.

2. The directory in which the index file and object module library for a modular compilation were to be placed is not defined correctly.

Your Response: 1. Create the index file by specifying the /INDEXFILE qualifier and the /CREATE qualifier in the command.

2. Define a directory to the logical name OPS$USERLIB.

%OPSCOMP-E-NOOPS5, No OPS5 source files specified

Explanation: A required VAX OPS5 source file was not specified in the command.

Your Response: Correct the command by specifying a VAX OPS5 source file.

%OPSCOMP-E-OPENIN, Unable to open DEV:FILNAM.TYP for input

Explanation: The input file DEV:FILNAM.TYP was not found.

Your Response: Correct the command by referring to an existing file or proper device.

%OPSCOMP-E-OPENINDEXIN, Unable to open file DEV:FILNAM.TYP for input

Explanation: 1. The /INDEX_FILE qualifier was specified in the command without a file name.

2. The directory in which the index file and object module library for a modular compilation were to be placed was not defined correctly.

Your Response: 1. Specify the /INDEX_FILE qualifier with a file name.

2. Check that the logical name OPS$USERLIB is defined to be the directory containing the modular compilation index file.

%OPSCOMP–E–OPENINDEXOUT, Unable to open file DEV:FILNAM.TYP for output

Explanation: 1. The /INDEX_FILE qualifier was specified in the command without a file name.

2. The directory in which the index file and object module library for a modular compilation were to be placed was not defined correctly.

Your Response: 1. Specify the /INDEX_FILE qualifier with a file name.

2. Check that the logical name OPS$USERLIB is defined to be the directory containing the modular compilation index file.

%OPSCOMP–E–PREMATURE_EOF, Premature end of file

Explanation: The compiler expected additional input.

Your Response: Check the source file for a missing closing parenthesis or missing code.

%OPSCOMP–E–SYNTAX_ERROR, One of the following symbols was expected: AAAAAA AAAAAA ...

Explanation: The compiler identified an incorrect symbol—syntax error—in the input file. The values AAAAAA are symbols that can replace the incorrect symbol. In a listing file, this message is usually followed with another message that identifies the compiler's recovery action.

Your Response: Check for a typing error, replace the incorrect symbol, and recompile the program.

%OPSCOMP–E–UNRECOVERABLE, Unrecoverable syntax error

Explanation: The compiler could not recover from a syntax error. This message is always preceded by the syntax error message.

Your Response: Check the source file for a typing error, replace the incorrect symbol, and recompile the program.

%OPSCOMP–F–DIRECTORY, Insufficient resources available or error in directory

Explanation: 1. You exceeded your disk quota.

2. Not enough room was available on the specified device to create the output file.

3. The default directory was set to a directory that could not be written to.

Your Response: 1. Use the DCL command PURGE to create additional disk space or ask the system manager to increase your disk quota.

2. Enlarge storage space by deleting unnecessary files from the output volume and transferring them to a backup volume. Use another volume with more space or use a larger volume—for example, from RX02 diskette to RL02.

3. Set the default directory to a directory that can be written to or change the default directory's protection.

%OPSCOMP-F-FATAL_ERROR, Fatal error—compilation aborted

Explanation: An unrecoverable error occurred. This message is always preceded by a message that gives the reason for the abort.

Your Response: Refer to the message that preceded this message and take the appropriate corrective action.

%OPSCOMP-F-INCOMPAT, OPS5 compiler not compatible with VMS version

Explanation: The version of the VMS operating system was not recent enough.

Your Response: Upgrade the VMS operating system to a more recent version, reinstall the VAX OPS5 compiler, and try again.

%OPSCOMP-F-LIZELIM, Maximum number of unique literalize entries exceeded

Explanation: The VAX OPS5 compiler's internal capacity for literalize entries was exceeded. The current limit on the number of literalize entries you can declare in a single program is 2048.

Your Response: Edit the source file to reflect the proper limit.

%OPSCOMP-F-LIZENOADD, Maximum number of literalize entries exceeded—AAAAAA not added

Explanation: The VAX OPS5 compiler could not declare the literalize entry AAAAAA because too many literalize entries were declared for the program. The current limit on the number of literalize entries you can declare in a single program is 2048.

Your Response: Edit the source file to reflect the proper limit.

%OPSCOMP-F-NOVIRMEM, Insufficient memory

Explanation: The compiler exhausted virtual memory.

Your Response: Ask the system manager to check that your process quota PGFLQUOTA and the value of the SYSGEN parameter VIRTUALPAGECNT are large enough.

%OPSCOMP-F-PUNT, Internal inconsistency—AAAAAA

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital, specifying AAAAAA from the message. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTADDANY, Internal inconsistency—ADD_ANY_TEST

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTLIZE, Internal inconsistency—ADD_TO_E_CLASS

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTOPCODESYN, Internal inconsistency—OPCODESYN

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTOPERANDS, Internal inconsistency—operand stack overflow

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTOPERATORS, Internal inconsistency—operator stack overflow

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTPRSSP, Internal inconsistency—parse stack overflow

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-F-PUNTRHSOVFL, Internal inconsistency—REMOVE_LIST

Explanation: The compiler detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSCOMP-I-ENDCOMPILE, End of compilation dd-mmm-yyyy hh:mm:ss

Explanation: The compilation completed. The message includes the date and time the compilation completed.

Your Response: None. The message is informational.

%OPSCOMP-I-LINESREAD, Compiled n lines

Explanation: The compiler processed n lines of source code.

Your Response: None. The message is informational.

%OPSCOMP-I-NOERRORS, No errors detected

Explanation: The compiler completed the compilation without detecting an error.

Your Response: None. The message is informational.

%OPSCOMP-I-NUMERRORS, n errors detected

Explanation: The compiler detected n errors.

Your Response: Use an editor to correct the errors in the source file and recompile it.

%OPSCOMP-I-NUMWARNINGS, n warnings detected

Explanation: The compiler detected n warnings.

Your Response: Use an editor to correct the errors in the source file and recompile it.

%OPSCOMP-I-OPENLISTOUT, Unable to open DEV:FILENAME.TYPE—/NOLIST assumed

Explanation: The /LIST qualifier was specified with an output file that could not be opened. The source file was compiled but no listing file was generated.

Your Response: None. The message is informational.

%OPSCOMP-I-RECOMPILE, Recompile module n

Explanation: This message is always preceded by a message that informs you that an attribute name was declared in more than one LITERAL declaration in the program. The message informs you which module contains the incompatible declaration.

Your Response: Specify a unique attribute name in the erroneous LITERAL declaration and then recompile the module.

%OPSCOMP-I-TIMEUSED, Time used was s.ss seconds

Explanation: The compiler used s.ss seconds to perform the specified compilation.

Your Response: None. The message is informational.

%OPSCOMP-W-ARGCOUNT, Number of arguments used does not match number declared

Explanation: You used a different number of arguments than you declared.

Your Response: Correct and recompile the module(s) concerned.

%OPSCOMP-W-BADINDEX, Index file DEV:FILNAM.TYP corrupted

Explanation: A modular compilation was performed and the resulting data base was corrupted.

Your Response: Recompile the modules.

%OPSCOMP-W-BADINDEXFILE, Conflicting values in index file for attribute name AAAAAA

Explanation: The attribute name AAAAAA was defined with two different values. An attribute name can have only one value in a program.

Your Response: Correct the appropriate module. If the program consists of more than one module, recompile the module you corrected specifying the /CREATE qualifier to rebuild the modular compilation index file. Then recompile the remaining modules without specifying the /CREATE qualifier.

%OPSCOMP–W–BADMODIFYCE, Invalid condition element number specified in MODIFY action: n

Explanation: A condition element was specified by a number that was greater than the number of positive condition elements in the production's left-hand side.

Your Response: Replace the number with a number that is less than or equal to the number of positive condition elements in the production's left-hand side. Alternatively, bind the condition element to an element variable on the production's left-hand side and replace the erroneous number with that variable.

%OPSCOMP–W–BADREMOVECE, Invalid condition element number specified in REMOVE action: n

Explanation: A condition element was specified by a number that was greater than the number of positive condition elements in the production's left-hand side.

Your Response: Replace the number with a number that is less than or equal to the number of positive condition elements in the production's left-hand side. Alternatively, bind the condition element to an element variable on the production's left-hand side and replace the erroneous number with that variable.

%OPSCOMP–W–BADSUBSTRCE, Invalid condition element number specified in call to SUBSTR function: n

Explanation: A condition element was specified by a number that was greater than the number of positive condition elements in the production's left-hand side.

Your Response: Replace the number with a number that is less than or equal to the number of positive condition elements in the production's left-hand side. Alternatively, bind the condition element to an element variable on the production's left-hand side and replace the erroneous number with that variable.

%OPSCOMP–W–BIGWATCH, Argument to WATCH greater than 4

Explanation: In the startup statement, the argument to WATCH was a value greater than 4.

Your Response: Respecify the argument to WATCH with a value between 0 and 4.

%OPSCOMP–W–CALLOLD, External routine result assumed to be in R0

Explanation: Your program called an external function expecting the result to be returned in R0, but the declaration of the function implies the result will be returned using OPS$VALUE.

Your Response: Check the external routine, its declaration in the program, and how it is called. For further information, see the *VAX OPS5 Reference Manual.*

%OPSCOMP-W-CANTLIT, Unable to assign literalize entries within n literal values—values assigned at random

Explanation: A LITERALIZE declaration contained n attribute names and n was greater than 256. (A working-memory element cannot have more than 256 atoms.)

Your Response: Rename some of the attributes in your program to conform to the maximum limit.

%OPSCOMP-W-DUPCATCHHERE, Duplicate production name or catcher name AAAAAA in this module—catcher ignored

Explanation: The symbol AAAAAA named more than one catcher in the module or named a catcher and a production in the module. The names of catchers and productions in a program must be unique.

Your Response: Correct the source file.

%OPSCOMP-W-DUPCATCHTHERE, Duplicate production name or catcher name AAAAAA in another module—catcher ignored

Explanation: The CATCH action was specified with the symbol AAAAAA, and that symbol named a production in a different module. The names of catchers and productions in a program must be unique.

Your Response: Use the DCL command SEARCH to search each source file for the duplicate name. When you find the duplicate name, change it.

%OPSCOMP-W-DUPDECLIT, Duplicate attribute name AAAAAA—second occurrence ignored

Explanation: The attribute name AAAAAA was specified more than once in a LITERALIZE declaration. All attribute names specified in a LITERALIZE declaration must be unique.

Your Response: Correct the source file.

%OPSCOMP-W-DUPDECRULE, Duplicate production name, AAAAAA—production ignored

Explanation: More than one production in the source file had the name AAAAAA. The names of the productions in a program must be unique.

Your Response: Correct the source file.

%OPSCOMP-W-DUPRULEHERE, Duplicate production name or catcher name AAAAAA in this module—production ignored

Explanation: The symbol AAAAAA named more than one production in the module or named a production and a catcher in the module. The names of catchers and productions in a program must be unique.

Your Response: Correct the source file.

%OPSCOMP-W-DUPRULETHERE, Duplicate production name or catcher name AAAAAA in another module—production ignored

Explanation: The source files for one or more modules contained productions that have the name AAAAAA. The names of the productions in a program must be unique.

Your Response: Use the DCL command SEARCH to search each source file for the duplicate name. When you find the duplicate name, change it.

%OPSCOMP-W-DUPWMECLASS, Duplicate LITERALIZE declaration for working-memory element class AAAAAA

Explanation: Two LITERALIZE declarations specify the same class name.

Your Response: Make sure each LITERALIZE declaration specifies a unique class name.

%OPSCOMP-W-EXTCALL, Subroutine AAAAAA not declared external

Explanation: The subroutine AAAAAA was specified in a right-hand side call action but was not previously declared with the EXTERNAL declaration.

Your Response: Declare the subroutine AAAAAA in an EXTERNAL declaration before using the subroutine in a right-hand side CALL action.

%OPSCOMP-W-EXTDUPL, Routine AAAAAA already declared. Declaration ignored

Explanation: External routine AAAAAA has already been declared in this module. The compiler uses the first declaration.

Your Response: Delete the second declaration.

%OPSCOMP-W-EXTFUNC, Function AAAAAA not declared external

Explanation: An attempt to call the external function AAAAAA failed because AAAAAA was not declared.

Your Response: Correct the source file.

%OPSCOMP-W-EXTGLOB, Routine AAAAAA declared in another module. Declaration ignored

Explanation: External routine AAAAAA has already been declared in a previous module. The compiler uses the first declaration.

Your Response: Delete the second declaration.

%OPSCOMP-W-ILLCOMPLIT, Literalize declaration incompatible with other modules

Explanation: A LITERALIZE declaration assigned to a working-memory element different attribute names than were assigned in another module. This message is always followed by a message that informs you which module to correct.

Your Response: Make sure all LITERALIZE declarations assign the same attribute names to working-memory elements that have the same class name.

%OPSCOMP-W-INC_EOL, Non-commentary text after %INCLUDE ignored

Explanation: The compiler found text that was not a comment following a %INCLUDE on the same line.

Your Response: Check the line.

%OPSCOMP-W-INCOMPLIT, Literal declaration AAAAAA = n incompatible with other modules

Explanation: More than one LITERAL declaration assigned a field to the attribute name AAAAAA. An attribute name can have only one field in a program.

Your Response: Correct the appropriate source file.

%OPSCOMP-W-INDEXEOF, Premature end of index file DEV:FILNAM.TYP

Explanation: A modular compilation failed because one or more of the modules was not complete.

Your Response: Check that all modules are complete and recompile them.

%OPSCOMP-W-INVLITVAL, Attribute name AAAAAA not declared

Explanation: The symbol AAAAAA was specified in a call to the LITVAL function and was not previously declared with the LITERAL, LITERALIZE, or VECTOR-ATTRIBUTE declaration.

Your Response: Declare the symbol.

%OPSCOMP-W-INVTEST, Invalid test on symbol

Explanation: The predicate greater-than (>), greater-than-equal-to (>=), less-than (<), or less-than-equal-to (<=) was used to test a symbol. These predicates must be used only with numbers or variables bound to numbers.

Your Response: Correct the source file.

%OPSCOMP-W-IVPRED, Invalid predicate with variable AAAAAA

Explanation: A predicate was used with the variable AAAAAA before the variable was bound to a value.

Your Response: Bind the variable to a value before you use the variable with a predicate other than the equal operator (=).

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

Explanation: A LITERALIZE declaration contains an attribute name that was previously assigned a field that does not match the attribute name's current field.

Your Response: Correct the source file.

%OPSCOMP–W–LITIZE1, Attribute name AAAAAA was previously assigned the value 1—this conflicts with LITERALIZE conventions

Explanation: The attribute name AAAAAA was specified in a LITERALIZE declaration and field 1 was previously assigned to that name. The LITERALIZE declaration always assigns field 1 to class name; therefore field 1 cannot be assigned to AAAAAA.

Your Response: Correct the LITERALIZE declaration by using a different attribute name.

%OPSCOMP–W–MECHBAD, Passing mechanism in declaration ignored

Explanation: The declaration of the external routine used in a CALL action specified an argument-passing mechanism.

Your Response: A CALL action passes all arguments in the result element. Check the external routine and the EXTERNAL declaration.

%OPSCOMP–W–NEGWATCH, Argument to WATCH less than 0

Explanation: In the startup statement, the argument to WATCH was a value less than 0.

Your Response: Respecify the argument to WATCH with a value between 0 and 4.

%OPSCOMP–W–NOCONCAT, File concatenation not allowed with modular compilation

Explanation: An attempt to perform a modular compilation failed because the command contained more than one source file.

Your Response: Compile each source file with a separate command.

%OPSCOMP–W–QUOTE, Atom missing closing quote

Explanation: An atom was specified with an opening quote but no closing quote.

Your Response: Place a closing quote after the atom.

%OPSCOMP–W–REASSGNLIT, Attempt to assign attribute name AAAAAA a new value—old value kept

Explanation: A LITERAL declaration failed to reassign a new field to the attribute name AAAAAA. The attribute name retained its old field.

Your Response: Correct the source file.

%OPSCOMP–W–RETIGNORE, Return value specified in declaration ignored

Explanation: The declaration of the external routine used in a CALL action specified a function return type.

Your Response: A CALL action does not expect any return type. Check the external routine and the EXTERNAL declaration.

%OPSCOMP–W–STARTTHERE, STARTUP already compiled from another module

Explanation: More than one module contains a STARTUP statement.

Your Response: Either merge the STARTUP statements or delete all but one of them.

%OPSCOMP-W-TOOMANYCES, Too many positive condition elements on left-hand side

Explanation: The left-hand side of a production has more positive condition elements than are allowed. The maximum number allowed is 32.

Your Response: Simplify your program so that no more than 32 positive condition elements are specified for the left-hand side of any production.

%OPSCOMP-W-TWOSTARTS, More than one STARTUP statement found

Explanation: The source file contained more than one STARTUP statement.

Your Response: Either merge the STARTUP statements or delete all but one of them.

%OPSCOMP-W-TWOVAS, Working-memory element class AAAAAA contains the names of two vector attributes

Explanation: A LITERALIZE declaration contains the name of more than one vector attribute.

Your Response: Correct the source file.

%OPSCOMP-W-TYPEBAD, Type used (AAAAAA) does not match declared type (BBBBBB)

Explanation: An external routine was called with an argument of type AAAAAA, but in the EXTERNAL declaration, the argument had type BBBBBB.

Your Response: Check the EXTERNAL declaration and how the routine is called.

%OPSCOMP-W-UNDECCEVAR, Element variable AAAAAA not bound

Explanation: The element variable AAAAAA was used in a MODIFY action, REMOVE action, or a call to the SUBSTR function, but was not previously bound to a working-memory element on the left-hand side of the production or in a right-hand side CBIND action.

Your Response: Bind the element variable to a working-memory element on the left-hand side of the production or in a right-hand side CBIND action.

%OPSCOMP-W-UNDECLIT, Attribute name AAAAAA not declared—assigned 1

Explanation: The source file contained the attribute name AAAAAA, which was not declared. The VAX OPS5 run-time system assigns field 1 to the attribute name.

Your Response: Make sure you typed the attribute name correctly and declared it in a LITERAL or LITERALIZE declaration.

%OPSCOMP-W-VARNOTBOUND, Variable AAAAAA not bound

Explanation: The variable AAAAAA was referred to on the right-hand side of a production and was not previously bound to a value on the left-hand side of the production or in a right-hand side BIND action.

Your Response: Bind the variable to a value on the left-hand side of the production or in a right-hand side BIND action.

%OPSCOMP-W-VERYBIGNUM, Size of number exceeds implementation limits

Explanation: The source file contained a number that exceeded the maximum number allowed.

Your Response: See the *VAX OPS5 Reference Manual* for the valid range for integers or for floating-point numbers and correct the source file.

%OPSRT-F-BADCBINDVAR, CBIND—too many CBIND actions

Explanation: The right-hand side of a production contained more CBIND actions than are allowed. The maximum number allowed is 64.

Your Response: Simplify your program so that no production has more than 64 CBIND actions on the right-hand side.

%OPSRT-F-BADCENUM, CBIND—invalid element designator: n

Explanation: The run-time system detected an internal inconsistency.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSRT-F-BADWRITE, Writing nonexistent file

Explanation: The run-time system tried to write to a file that did not exist.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSRT-F-CREZONE, Code inconsistency attempting to create memory zone

Explanation: A problem occurred when the run-time system tried to create a memory zone.

Your Response: Ask the system manager to check that your process quota PGFLQUOTA and the value of the SYSGEN parameter VIRTUALPAGECNT are both large enough.

%OPSRT-F-FREEMEM, Code inconsistency attempting to free virtual memory

Explanation: A problem occurred when the run-time system tried to free virtual memory.

Your Response: Ask the system manager to check that your process quota PGFLQUOTA and the value of the SYSGEN parameter VIRTUALPAGECNT are both large enough.

%OPSRT–F–GETMEM, Code inconsistency attempting to get virtual memory

Explanation: A problem occurred when the run-time system tried to get more virtual memory.

Your Response: Ask the system manager to check that your process quota PGFLQUOTA and the value of the SYSGEN parameter VIRTUALPAGECNT are both large enough.

%OPSRT–F–INSVIRMEM, Insufficient virtual memory

Explanation: The run-time system exhausted virtual memory.

Your Response: Ask the system manager to check that your process quota PGFLQUOTA and the value of the SYSGEN parameter VIRTUALPAGECNT are both large enough.

%OPSRT–F–MISMATCH, Incompatible versions

Explanation: The versions of the run-time system and the compiler are not the same.

Your Response: Use the DCL command ANALYZE/IMAGE to make sure the versions of the run-time system and compiler are the same, or reinstall VAX OPS5.

%OPSRT–I–BREAKNOTED, BREAK—break

Explanation: The VAX OPS5 run-time system encountered a breakpoint set by the PBREAK or WBREAK command.

Your Response: None. The message is informational.

%OPSRT–I–CANCELED, OPS$CANCEL_RUN used

Explanation: The OPS$CANCEL_RUN support routine was used by your program.

Your Response: None. The message is informational.

%OPSRT–I–CTRLCNOTED, CTRL/C—return to command interpreter

Explanation: A Ctrl/C was typed during program execution. Typing Ctrl/C invokes the command interpreter.

Your Response: The message is informational. Respond to the command interpreter prompt with a command.

%OPSRT–I–EMPTYCS, HALT—no satisfied productions

Explanation: Execution of the program halted because the conflict set was empty.

Your Response: None. The message is informational.

%OPSRT–I–EXIT, User entered EXIT or CTRL/Z

Explanation: The EXIT or Ctrl/Z command was entered by the user.

Your Response: None. The message is informational.

%OPSRT–I–HALTED, HALT—right-hand-side action

Explanation: Execution of the program halted because the HALT action was executed.

Your Response: None. The message is informational.

%OPSRT–I–NORMAL, Normal, successful completion

Explanation: The program completed normally without errors.

Your Response: None. The message is informational.

%OPSRT–I–NOSTARTUP, No STARTUP code in program

Explanation: The program does not include a STARTUP statement.

Your Response: The message is informational. You may include a STARTUP statement and recompile the program, if you wish.

%OPSRT–I–PAUSED, Pause

Explanation: The VAX OPS5 run-time system paused program execution because the RUN command was specified with an integer n, and n recognize-act cycles were processed. The command interpreter was invoked.

Your Response: The message is informational. Respond to the prompt with a command.

%OPSRT–I–PBREAK, PBREAK encountered

Explanation: The VAX OPS5 run-time system encountered a breakpoint set by the PBREAK command.

Your Response: The message is informational. Respond to the prompt with a command.

%OPSRT–I–WBREAK, WBREAK encountered

Explanation: The VAX OPS5 run-time system encountered a breakpoint set by the WBREAK command.

Your Response: The message is informational. Respond to the prompt with a command.

%OPSRT–W–ACCEPTARGS, ACCEPT—too many arguments

Explanation: More than one argument was specified with the ACCEPT function.

Your Response: Delete the extra arguments.

%OPSRT–W–ACCEPTFILE, ACCEPT—file not open for input

Explanation: The file identifier specified with the ACCEPT function was not associated with an open input file.

Your Response: Check for a typing error in the file identifier. If no typing error exists, use the OPENFILE action or command to open the appropriate file for input and associate the file with a file identifier.

%OPSRT-W-ADDFILE, ADDSTATE—error in file processing

Explanation: The VAX OPS5 run-time system was unable to open the file specified with the ADDSTATE action or command.

Your Response: Make sure the file specification used with the ADDSTATE action or command is a valid VMS file specification.

%OPSRT-W-AFTERUSAGE, AFTER—invalid format

Explanation: The AFTER action or command was not specified in the correct format.

Your Response: See the *VAX OPS5 Reference Manual* for a description of the correct syntax format and correct the source file or the command.

%OPSRT-W-ALPHATOKEN, Deleting nonexistent token

Explanation: An internal run-time error occurred.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSRT-W-BACKDISABLD, BACK—command disabled

Explanation: An attempt was made to use the BACK command and the command was disabled.

Your Response: The BACK command is disabled by default. Use the ENABLE command to enable the BACK command before you use it.

%OPSRT-W-BACKLIMIS, BACK—the current limit is n

Explanation: The argument value specified with the BACK command exceeded the limit of recognize-act cycles over which you can back up. The maximum number is 64 cycles.

Your Response: Specify an integer less than or equal to 64 as the argument value to the BACK command.

%OPSRT-W-BACKPOS, BACK—argument value is negative

Explanation: The argument value specified with the BACK command was negative.

Your Response: Specify the BACK command with a positive integer.

%OPSRT-W-BACKTOOFAR, BACK—argument value exceeds limit

Explanation: The argument value specified with the BACK command exceeded the limit of recognize-act cycles over which you can back up. The maximum number is 64 cycles.

Your Response: Specify an integer less than or equal to 64 as the argument value for the BACK command.

%OPSRT-W-BADARGNUM, Wrong number of arguments

Explanation: The DEFAULT action or command was not specified with two arguments, or the OPENFILE action or command was not specified with three arguments.

Your Response: Specify the DEFAULT action or command with two arguments: a file identifier and the keyword ACCEPT, TRACE, or WRITE. Specify the OPENFILE action or command with three arguments: a file identifier, VMS file specification, and the keyword IN, OUT, or APPEND.

%OPSRT-W-BADCOMPUTE, COMPUTE—symbol used in arithmetic expression: AAAAAA

Explanation: The COMPUTE function or the routine OPS$CVAN was specified with a symbol or a variable bound to a symbol.

Your Response: Check for a typing error. Specify the COMPUTE function with numbers or variables bound to numbers.

%OPSRT-W-BADCVNA, OPS$CVNA—overflow while converting an integer to an atom

Explanation: An integer was converted to a numeric atom, and the atom did not fit in 30 bits (32 bits minus two tag bits) of storage space.

Your Response: Make sure the integer converts to a numeric atom that fits in 30 bits of storage space. That is, a numeric atom must be in the range of –(2**30–1) to (2**30–1).

%OPSRT-W-BADFILESPEC, OPENFILE—invalid file specification: DEV:FILNAM.TYP

Explanation: The VAX OPS5 run-time system could not open the file specified with the OPENFILE action or command.

Your Response: Check for a typing error. The second argument value specified with the OPENFILE action or command must be a symbol that represents a valid VMS file specification.

%OPSRT-W-BADMODE, OPENFILE—argument value not IN, OUT or APPEND: AAAAAA

Explanation: The third argument value specified with the OPENFILE action or command was a keyword other than IN, OUT, or APPEND.

Your Response: Check for a typing error. Specify the OPENFILE action or command with the keyword IN, OUT, or APPEND.

%OPSRT-W-BADMODIFYPOS, MODIFY—attempted to change an atom in a nonexistent field

Explanation: 1. The maximum number of atoms a working-memory element can have is 383. This limit was exceeded.

2. An attempt was made to create a vector value that uses more than the allowable number of fields.

3. An invalid value was specified with the OPS$TAB support routine.

Your Response: Correct the source file so that no working-memory element has more than 383 atoms.

%OPSRT-W-BADREAD, Reading nonexistent file

Explanation: The VAX OPS5 run-time system tried to read a file that did not exist.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSRT-W-BADTABVAL, PPWM—attribute name not declared: AAAAAA

Explanation: The PPWM command was specified with an attribute whose name was not declared.

Your Response: Check for a typing error and reenter the command. If no typing error exists, declare the attribute name in the source file and then recompile the file.

%OPSRT-W-BETATOKEN, Deleting nonexistent token

Explanation: An internal run-time error occurred.

Your Response: Submit an SPR to Digital. Include with the SPR a program listing and a machine-readable source program, if possible.

%OPSRT-W-CANTOPEN, OPENFILE—unable to open file: DEV:FILNAM.TYP

Explanation: The VAX OPS5 run-time system could not open the file specified with the OPENFILE action or command.

Your Response: Check for a typing error. Check that the file exists and that you have the appropriate access rights to it.

%OPSRT-W-CLOSEFILEID, CLOSEFILE—file-identification name not a symbol: n

Explanation: The file identifier specified with the CLOSEFILE action or command was not a symbol.

Your Response: Specify the CLOSEFILE action or command with a file identifier that is a symbol.

%OPSRT-W-CSARG, CS—arguments specified

Explanation: The CS command was specified with an argument.

Your Response: Reenter the CS command with no arguments.

%OPSRT-W-CYCLEOFLOW, Cycle count overflow. The cycle count will be reset

Explanation: The number of recognize-act cycles overflowed the 32-bit integer limit of the counter. This overflow temporarily disables the BACK command.

Your Response: None. The counter will reinitialize itself.

%OPSRT-W-DEFAULTARGS, DEFAULT—wrong number of arguments

Explanation: The DEFAULT action or command was not specified with two arguments.

Your Response: Specify the DEFAULT action or command with two arguments: a location and a keyword.

%OPSRT-W-DEFAULTFILEID, DEFAULT—file-identification name not a symbol: n

Explanation: The file identifier specified with the DEFAULT action or command was not a symbol.

Your Response: Specify the DEFAULT action or command with a file identifier that is a symbol.

%OPSRT-W-DEFAULTIN, DEFAULT—file not open for input: AAAAAA

Explanation: The DEFAULT action or command was specified with the ACCEPT keyword and a symbol that was not a file identifier for an open input file.

Your Response: Use the OPENFILE action or command to open the appropriate file for input and associate the file with a file identifier. Then specify the DEFAULT action or command with that file identifier and the keyword ACCEPT.

%OPSRT-W-DEFAULTOUT, DEFAULT—file not open for output: AAAAAA

Explanation: The DEFAULT action or command was specified with the TRACE or WRITE keyword and a symbol that was not a file identifier for an open output file.

Your Response: Use the OPENFILE action or command to open the appropriate file for output and associate the file with a file identifier. Then specify the DEFAULT action or command with that file identifier and the keyword TRACE or WRITE.

%OPSRT-W-DEFAULTUSE, DEFAULT—argument value not ACCEPT, TRACE, or WRITE: AAAAAA

Explanation: The second argument used with the DEFAULT action or command was not the ACCEPT, TRACE, or WRITE keyword.

Your Response: Use the keyword ACCEPT, TRACE, or WRITE as the second argument when you use the DEFAULT action or command.

%OPSRT-W-DEFIN, ACCEPT—default file not open for input

Explanation: An attempt was made to read a file that was not open for input.

Your Response: Open the file prior to the Read action.

%OPSRT-W-DEFOUT, WRITE—default file not open for output

Explanation: An attempt was made to write to a file that was not open for output.

Your Response: Open the file prior to the Write action.

%OPSRT-W-DEFTRACE, TRACE—default file not open for output

Explanation: An attempt was made to write trace output to a file that was not open for output.

Your Response: Open the file prior to the Write action. If a file is not open to receive trace output, the data is sent to the user's terminal screen.

%OPSRT-W-DELWME, Attempted to use a WME that has been deleted

Explanation: The working-memory element your program tried to use has already been deleted.

Your Response: Correct and recompile the program module(s).

%OPSRT-W-FILEIDINUSE, OPENFILE—file-identification name already in use: AAAAAA

Explanation: The file identifier specified with the OPENFILE action or command was not unique.

Your Response: Make sure the file identifier you specify with the OPENFILE action or command has not been assigned to a file in a previous use of the OPENFILE action or command.

%OPSRT-W-ILLARG, —invalid argument value: AAAAAA

Explanation: The argument value specified with a command was not valid.

Your Response: Reenter the command with a valid argument value.

%OPSRT-W-ILLCMD, Invalid command: AAAAAA

Explanation: The VAX OPS5 command interpreter did not recognize the command entered.

Your Response: Check for a typing error and enter a valid VAX OPS5 command.

%OPSRT-W-ILLSTRATEGY, STRATEGY—argument value not LEX or MEA: AAAAAA

Explanation: The STRATEGY command was specified with a keyword other than LEX or MEA.

Your Response: Check for a typing error and reenter the command with a correct argument value.

%OPSRT-W-ILLTAB, Attribute operator (^) specified

Explanation: The attribute operator (^) was specified with the ADDSTATE, RESTORESTATE, or SAVESTATE action or command.

Your Response: Check for a typing error and correct the source file.

%OPSRT-W-LITVALARG, LITVAL—attribute name not declared: AAAAAA

Explanation: The LITVAL function was specified with an attribute name that was not declared with the LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration.

Your Response: Check for a typing error. If no typing error exists, declare the attribute name with the LITERAL, LITERALIZE, or VECTOR–ATTRIBUTE declaration.

%OPSRT-W-NEGARG, Argument value is negative: –n

Explanation: The RUN command was specified with a zero or a negative integer.

Your Response: Reenter the RUN command without an argument or with a positive integer.

%OPSRT-W-NILFILEID, OPENFILE—NIL specified as file-identification name

Explanation: NIL was specified as a file identifier in a call to the OPENFILE action or command.

Your Response: Specify a symbol other than NIL for the file identifier.

%OPSRT-W-NOARGS, No arguments specified

Explanation: The @, ADDSTATE, CALL, DISABLE, ENABLE, MAKE, RESTORESTATE, or SAVESTATE command was specified without an argument.

Your Response: Reenter the command and specify the appropriate arguments.

%OPSRT-W-NOBUILDCOMP, BUILD—construct not built

Explanation: The construct was not built due to a previously reported error.

Your Response: Correct the error previously reported and try the BUILD again.

%OPSRT-W-NOINI, Already initialized

Explanation: The program called OPS$INITIALIZE more than once.

Your Response: Check the calling program.

%OPSRT-W-NONNUMARG, Argument value is a symbol

Explanation: In a RUN or BACK command, a symbolic argument was entered instead of a numeric argument.

Your Response: Reenter the command with a numeric argument.

%OPSRT-W-NORUN, OPS$RUN is not active

Explanation: The OPS$RUN support routine is not active.

Your Response: Check the calling program.

%OPSRT-W-NOSUCHWME, No such WME

Explanation: Your program tried to use a working-memory element that does not exist.

Your Response: Correct and recompile the relevant program module(s).

%OPSRT-W-NOTABVAL, No value after attribute operator (^)

Explanation: The CALL, MAKE, or PPWM command was specified with an attribute that was not followed by a value.

Your Response: Reenter the command and specify a value after the attribute.

%OPSRT-W-NOTACATCH, AFTER—argument value not a catcher: AAAAAA

Explanation: The argument value specified with the AFTER action or command was not the name of a catcher.

Your Response: Specify the AFTER action or command with the name of a catcher.

%OPSRT-W-NOTARULE, Argument value not a production name: AAAAAA

Explanation: The argument value specified with the MATCHES or EXCISE command was not the name of a production.

Your Response: Reenter the command with the name of a production.

%OPSRT-W-NOTEXTERNAL, CALL—routine not declared external: AAAAAA

Explanation: The subroutine AAAAAA was specified in a right-hand side call action but was not previously declared with the EXTERNAL declaration.

Your Response: Declare the subroutine AAAAAA in an external declaration before using the subroutine in a right-hand side CALL action.

%OPSRT-W-NOTINBUILD, Not executing a BUILD command

Explanation: ENDBUILD was specified without a BUILD action.

Your Response: Remove ENDBUILD. It should only be specified to end a BUILD action.

%OPSRT-W-NOTINI, Not initialized

Explanation: The program has not called the OPS$INITIALIZE support routine.

Your Response: OPS$INITIALIZE should be called before any other support routines. This only applies if the VAX OPS5 routine was compiled with /NOENTRY.

%OPSRT-W-NOTKEYWRD, ENABLE/DISABLE—argument value not BACK, HALT, TIMING or WARNING

Explanation: The ENABLE or DISABLE command was specified with an argument value other than BACK, HALT, TIMING, or WARNING.

Your Response: Check for a typing error and reenter the command with a correct argument value.

%OPSRT-W-OFLOWBACK, You cannot BACK over this point

Explanation: The number of recognize-act cycles overflowed the 32-bit integer limit of the counter. This overflow temporarily disables the BACK command.

Your Response: None. The counter will reinitialize itself.

%OPSRT-W-OPENARGS, OPENFILE—wrong number of arguments

Explanation: The OPENFILE action was not specified with three arguments.

Your Response: Make sure the OPENFILE action is specified with the correct number of arguments.

%OPSRT-W-OPENERR, @—unable to open specified file

Explanation: The specified input file could not be opened for the @ command.

Your Response: Make sure the file exists and is not protected against Read access.

%OPSRT-W-OPENFILEID, OPENFILE—file-identification name not a symbol: AAAAAA

Explanation: The file identifier specified with the OPENFILE action or command was not a symbol.

Your Response: The file identifier you specify with the OPENFILE action or command must be a symbol.

%OPSRT-W-OUTRANGEARG, WATCH—argument value outside valid range

Explanation: A value greater than 4 or less than 0 was specified as an argument to WATCH.

Your Response: Correct the program to place the WATCH argument value within the range of 0 to 4.

%OPSRT-W-PARAMARG, OPS$PARAMETER—invalid argument: n

Explanation: The argument specified with the OPS$PARAMETER routine did not point to a field in the result element.

Your Response: The maximum number of atoms the result element can have is 383. Make sure that the value specified with the OPS$PARAMETER routine does not exceed this limit.

%OPSRT-W-PMEDISABLED, REPORT—timing was never enabled

Explanation: The keyword TIMING was not specified in the REPORT command line.

Your Response: Reenter the REPORT command with the keyword TIMING specified.

OPSRT-W-PMEKEYWORD, REPORT—argument value not TIMING

Explanation: A keyword other than TIMING was specified in the REPORT command line. (TIMING is the only valid keyword for the REPORT command.)

Your Response: Reenter the REPORT command with the proper keyword—TIMING.

%OPSRT-W-RESTOREOF, RESTORESTATE—premature end of file

Explanation: Information, such as conflict set information and working-memory elements, was missing from the end of the file specified with the RESTORESTATE action or command.

Your Response: Make sure the file specified with the RESTORESTATE action or command is a file created with the SAVESTATE action or command and the file has not been edited. If the file was created and edited, use SAVESTATE to create the file again.

%OPSRT-W-RESTORERR, RESTORESTATE—error in file processing

Explanation: The VAX OPS5 run-time system could not open the file specified with the RESTORESTATE action or command.

Your Response: Check for a typing error. The argument value specified with the RESTORESTATE action or command must be a valid VMS file specification.

%OPSRT-W-RHSBADELEM, MODIFY—referred to nonexistent element

Explanation: An attempt was made to modify a working-memory element that does not exist.

Your Response: Check that the arguments specified with the MODIFY action or command are correct.

%OPSRT-W-RJCRLF, WRITE—CRLF cannot follow RJUST

Explanation: A call to the CRLF function immediately follows a call to the RJUST function.

Your Response: Correct the source file.

%OPSRT-W-RJNUMBER, RJUST—argument value is a symbol: AAAAAA

Explanation: The RJUST function was specified with a symbol.

Your Response: Specify the RJUST function with a positive integer.

%OPSRT-W-RJPOSITIVE, RJUST—argument value is less than or equal to 0: n

Explanation: The RJUST function was specified with 0 or a negative integer.

Your Response: Specify the RJUST function with a positive integer.

%OPSRT-W-RJRJ, WRITE—RJUST cannot follow RJUST

Explanation: A call to the RJUST function immediately follows another call to the RJUST function.

Your Response: Correct the source file.

%OPSRT-W-RJTABTO, WRITE—TABTO cannot follow RJUST

Explanation: A call to the TABTO function immediately follows another call to the RJUST function.

Your Response: Correct the source file.

%OPSRT-W-RUNNING, OPS5 system called recursively

Explanation: The OPS$RUN support routine is running and has been called again.

Your Response: Check the calling program.

%OPSRT-W-SAVEDWMES, RESTORESTATE—too many working-memory elements in saved file

Explanation: The RESTORESTATE action or command tried to create an instantiation that contained too many time tags.

Your Response: Make sure the file specified with the RESTORESTATE action or command is a file created with the SAVESTATE action or command and the file has not been edited. If the file was created and edited, use SAVESTATE to create the file again.

%OPSRT-W-SAVEFILE, SAVESTATE—error in file processing

Explanation: The VAX OPS5 run-time system could not open the file specified with the SAVESTATE action or command.

Your Response: Check for a typing error. The argument value specified with the SAVESTATE action or command must be a valid VMS file specification.

%OPSRT-W-SAVEVERSION, RESTORESTATE or ADDSTATE used old SAVESTATE file which will not be supported after this release

Explanation: The savestate file used by a RESTORESTATE or ADDSTATE operation is in an old format.

Your Response: Create the SAVESTATE file again.

%OPSRT-W-SHOKEYWORD, SHOW—argument value not SPACE

Explanation: An argument other than SPACE was specified in the SHOW command line.

Your Response: Reenter the SHOW command with the argument SPACE.

%OPSRT-W-STRINGSIZE, WRITE—string is too long for output buffer

Explanation: An attempt was made to write to an output file whose buffer was not large enough to store the string. The maximum buffer size allowed for storing a string is 2048 characters.

Your Response: Use the CRLF function to break the string into smaller pieces that fit the buffer.

%OPSRT-W-SUBSTRARG, SUBSTR—unbound argument value: AAAAAA

Explanation: The value of the second or third argument specified with the SUBSTR function was an attribute name that was not declared with the LITERAL, LITERALIZE, or VECTOR-ATTRIBUTE declaration.

Your Response: Check for a typing error. If no typing error exists, declare the attribute name with the LITERAL, LITERALIZE, or VECTOR-ATTRIBUTE declaration.

%OPSRT-W-SUBSTRINDEX, SUBSTR—second or third argument value is less than 1: n

Explanation: The value of the second or third argument specified with the SUBSTR function was an integer less than one. The second and third arguments point to fields in an element.

Your Response: Specify integers greater than or equal to one for the second and third arguments to the SUBSTR function.

%OPSRT–W–TABTONUMBER, TABTO—argument value is a symbol: AAAAAA

Explanation: The TABTO function was specified with a symbol or a variable bound to a symbol.

Your Response: Check for a typing error. Specify the TABTO function with an integer or a variable bound to an integer that is greater than or equal to one.

%OPSRT–W–TABTOPOSITIVE, TABTO—argument value is less than or equal to 0: n

Explanation: The TABTO function was specified with 0 or a negative integer.

Your Response: Specify the TABTO function with a positive integer.

%OPSRT–W–TAGOFLOW, Time tag overflow. WMEs will be reassigned new time tags in the original order

Explanation: The WME time-tag counter has overflowed. All WMEs in working memory when this occurs are given new time tags, however, their order of recency is preserved.

Your Response: None.

%OPSRT–W–TOOMANYARGS, Too many arguments specified

Explanation: More than one argument was specified with the @, ADDSTATE, DISABLE, ENABLE, RESTORESTATE, RUN, SAVESTATE, or STRATEGY command.

Your Response: Specify only one argument with the @, ADDSTATE, DISABLE, ENABLE, RESTORESTATE, RUN, SAVESTATE, or STRATEGY command.

%OPSRT–W–TOOMANYCHARS, Too many characters in atom

Explanation: The number of characters in an atom's print name exceeded the maximum limit of 256.

Your Response: Rename the atom so that the number of characters is valid.

%OPSRT–W–TOOMANYPBREAKS, PBREAK—maximum number of breakpoints exceeded

Explanation: The number of breakpoints set at a given time exceeded the maximum limit of 8.

Your Response: Delete as many breakpoints as necessary to keep within the maximum limit.

%OPSRT–W–TYPEBAD, Atom is not of the required type

Explanation: The atom used is of a different type than declared.

Your Response: Correct and recompile the program module(s).

%OPSRT-W-UNBOUNDTAB, Attribute name not declared: AAAAAA

Explanation: The source file contains an attribute name that was not declared with the LITERAL, LITERALIZE, or VECTOR-ATTRIBUTE declaration.

Your Response: Check for a typing error in the name. If there is no error, declare the attribute name with the LITERAL, LITERALIZE, or VECTOR-ATTRIBUTE declaration.

%OPSRT-W-WRITENOTHING, WRITE—no arguments specified

Explanation: The WRITE action was specified without arguments.

Your Response: Specify the WRITE action with at least one argument.

;* Scanner advanced to "AAAAAA" *****

Explanation: The compiler recovered from a syntax error by stepping past input until it read the symbol AAAAAA. This message appears only in a listing file and is associated with a previous syntax error message.

Your Response: The message is informational.

;* Symbol deleted *****

Explanation: The compiler recovered from a syntax error by deleting an erroneous symbol. This message appears only in a listing file and is associated with a previous syntax error message.

Your Response: The message is informational.

;* Symbol replaced by "AAAAAA" *****

Explanation: The compiler recovered from a syntax error by replacing an erroneous symbol. This message appears only in a listing file and is associated with a previous syntax error message.

Your Response: The message is informational.

A.4.1 Product License Messages

The following messages are displayed if you attempt to use OPS5 without a compiler license.

%LICENSE-F-NOAUTH, DEC OPS5, use is not authorized on this node

-LICENSE-F-NOLICENSE, no license is active for this software product

-LICENSE-I-SYSMGR, please see your system manager

Explanation: An attempt has been made to access the OPS5 compiler on an unlicensed node. The action has failed.

Your Response: See the system manager about obtaining the proper license.

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
User's Guide
AA–EZ18C–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: ______________________________

____

My additional comments or suggestions for improving this manual:

____

____

I found the following errors in this manual:

Page	Description
______	______________________________
______	______________________________
______	______________________________

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