MACRO–64 Assembler for OpenVMS AXP Systems

Reference Manual

Order Number: AA–PT9KB–TE

November 1993

This manual describes the MACRO–64 assembly language.

Revision/Update Information:	This manual supersedes the *MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual* (Order Number AA–PT9KA–TE).
Software Version:	MACRO–64 Assembler for OpenVMS AXP Systems Version 1.1
Operating System and Version:	OpenVMS AXP Version 1.5 or higher

Digital Equipment Corporation
Maynard, Massachusetts

First printing, November 1992
Revised, November 1993

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

© Digital Equipment Corporation 1992, 1993. All rights reserved.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation: Alpha AXP, AXP, DEC, DECmigrate, DECwindows, Digital, OpenVMS, VAX, VAX MACRO, VMS, and the DIGITAL logo.

The following is a third-party trademark:

OSF/1 is a registered trademark of the Open Software Foundation, Inc.

ZK6338

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface xi

1 Introduction

1.1 Language Features 1–1
1.1.1 Alpha AXP Native-Mode Instructions 1–1
1.1.2 Direct Assignment Statements 1–1
1.1.3 Assembler Directives 1–1
1.1.3.1 General Assembler Directives 1–1
1.1.3.2 Macro Directives 1–2
1.2 Invoking MACRO–64 on OpenVMS Systems 1–2
1.2.1 Command-Line Qualifiers 1–2

2 Components of MACRO–64 Source Statements

2.1 Source Statement Format 2–1
2.1.1 Label Field 2–2
2.1.2 Operator Field 2–3
2.1.3 Operand Field 2–3
2.1.4 Comment Field 2–4
2.2 Character Set 2–4
2.3 Numbers 2–5
2.3.1 Integers 2–6
2.3.2 Floating-Point Numbers 2–6
2.4 Quoted Literals 2–7
2.5 Symbols 2–7
2.5.1 Permanent Symbols 2–7
2.5.2 Predefined Symbols 2–7
2.5.3 User-Defined Symbols and Macro Names 2–8
2.5.4 Determining Symbol Values 2–9
2.5.4.1 Using Symbols in the Operator Field 2–9
2.5.4.2 Using Symbols in the Operand Field 2–10
2.6 Temporary Labels Within Source Code 2–10
2.7 Label Addresses 2–12
2.7.1 Label Addresses, Optimization, and Code Alignment 2–12
2.7.2 Label Addresses and Automatic Data Alignment 2–13
2.8 Terms and Expressions 2–14
2.9 Unary Operators for Terms and Expressions 2–17
2.9.1 Radix Control Operators 2–17
2.9.2 Numeric Complement Operator 2–18
2.10 Binary Operators 2–18
2.10.1 Arithmetic Shift Operator 2–19
2.10.2 Logical AND Operator 2–19
2.10.3 Logical Inclusive OR Operator 2–19

2.10.4 Logical Exclusive OR Operator 2–19
2.11 Direct Assignment Statements 2–19
2.12 Current Location Counter 2–20

3 MACRO–64 Lexical Operators

3.1 Processing with Lexical Operators 3–1
3.2 Lexical Operator Syntax 3–1
3.3 Numeric Symbols and Lexical String Symbols 3–3
3.4 Lexical Substitution Operator 3–3
3.5 Lexical Escape Operator 3–3
3.6 Using Lexical Operators 3–5
3.7 Lexical Operators 3–7
%EDIT 3–8
%ELEMENT 3–9
%EXTRACT 3–10
%FREG 3–11
%INTEGER 3–12
%IREG 3–13
%LENGTH 3–14
%LOCATE 3–15
%REPEAT 3–16
%STRING 3–17
%TIME 3–18
%TYPE 3–19

4 Macro Arguments

4.1 Using Macro Arguments 4–1
4.2 Using Default Values 4–2
4.3 Using Keyword Arguments 4–2
4.4 Using String Arguments 4–3
4.5 Argument Concatenation 4–6
4.6 Passing Numeric Values of Symbols 4–7
4.7 Using Created Temporary Labels 4–7

5 MACRO–64 Assembler Directives

5.1 Program Sections 5–1
5.2 Automatic Data Alignment 5–2
5.3 Directives 5–3
.ADDRESS 5–6
.ALIGN 5–7
.ASCIC 5–9
.ASCID 5–10
.ASCII 5–11
.ASCIZ 5–12
.BASE 5–13
.BEGIN_EXACT 5–16
.BLK*x* 5–18
.BYTE 5–20

.CODE_ADDRESS 5–21
.D_FLOATING 5–22
.DEFINE_FREG 5–23
.DEFINE_IREG 5–25
.DISABLE 5–26
.ELSE 5–27
.ENABLE 5–28
.END 5–34
.ENDC 5–35
.ENDM 5–36
.ENDR 5–37
.END_EXACT 5–38
.ERROR 5–39
.EVEN 5–40
.EXTERNAL 5–41
.F_FLOATING 5–42
.G_FLOATING 5–43
.IDENT 5–44
.IF 5–45
.IF_x 5–49
.IIF 5–52
.INCLUDE 5–53
.INSTRUCTION 5–54
.IRP 5–55
.IRPC 5–57
.LIBRARY 5–59
.LINKAGE_PAIR 5–61
.LIST 5–62
.LOCAL_CODE_ADDRESS 5–63
.LOCAL_LINKAGE_PAIR 5–64
.LOCAL_PROCEDURE_DESCRIPTOR 5–65
.LONG 5–66
.MACRO 5–67
.MCALL 5–70
.MDELETE 5–71
.MEXIT 5–72
.NARG 5–74
.NCHR 5–75
.NLIST 5–76
.NOSHOW 5–77
.OCTA 5–78
.ODD 5–79
.PACKED 5–80
.PAGE 5–81
.PRINT 5–82
.PROCEDURE_DESCRIPTOR 5–83
.PSECT 5–84

.QUAD 5–88
.REPEAT 5–89
.RESTORE_PSECT 5–91
.SAVE_PSECT 5–92
.S_FLOATING 5–93
.SHOW 5–94
.SIGNED_BYTE 5–96
.SIGNED_WORD 5–97
.SUBTITLE 5–98
.T_FLOATING 5–99
.TITLE 5–100
.UNDEFINE_REG 5–101
.WARN 5–102
.WEAK 5–103
.WORD 5–104

6 MACRO–64 Supplied Macros

6.1 MACRO–64 Supplied Library 6–1
6.2 Routines and Lexical Scope 6–1
6.2.1 Routines and Program Sections 6–2
6.3 Using Macros to Control Program Sections 6–2
6.3.1 Defining Program Sections 6–2
6.3.2 Using Macro-Defined Symbols 6–3
6.3.3 Defining Procedure Type 6–4
6.3.4 Using Macros in Prologue Sections 6–4
6.3.5 Using Macros in Epilogue Sections 6–5
6.4 Programming Examples Using Supplied Macros 6–5
6.5 Using the $CALL Macro 6–6
6.5.1 Using $CALL in Source Code 6–7
6.6 Programming Considerations 6–7
6.6.1 Making Multiple Calls From the Same Routine 6–8
6.6.2 Nonstandard Linkage 6–8
6.6.3 Routine Restrictions 6–8
6.7 Macro Descriptions and Syntax Rules 6–8
$BEGIN_EPILOGUE 6–9
$CALL 6–10
$CODE_SECTION 6–15
$DATA_SECTION 6–16
$END_EPILOGUE 6–17
$END_PROLOGUE 6–18
$END_ROUTINE 6–19
$LINKAGE_PAIR 6–20
$LINKAGE_SECTION 6–21
$OPDEF 6–22
.PACKED 6–26
$PROCEDURE_DESCRIPTOR 6–28
$RESET_LP_LIST 6–29
$RETURN 6–30
$ROUTINE 6–31

A MACRO–64 Alpha AXP Architecture Quick Reference

A.1 Register Usage Conventions A–4
A.2 Instruction Operand Notation A–4
A.3 Instruction Qualifier Notation A–5
A.4 F-P Control Register (FPCR) Format A–5
A.5 Decodable Pseudo-Operations A–6
A.6 Common Architecture Opcodes in Numerical Order A–6
A.7 OpenVMS PALcode Instruction Summary A–11
A.8 PALcode Opcodes in Numerical Order A–13
A.9 Common Architecture Instructions A–15

B Programming with MACRO–64

B.1 MACRO-64 Programming Hints B–1
B.1.1 STARLET.MLB Migration Macros B–2
B.1.2 Integer Division B–2
B.1.3 User-Defined Register Symbols and Macros B–2
B.2 The OpenVMS Calling Standard B–2
B.2.1 Effects on Assembly-Time when Using Calling-Standard Macros B–3
B.3 Accessing Memory with Base Registers B–4
B.4 Types of Data Structures B–8
B.4.1 Procedure Descriptor B–8
B.4.2 Signature Block B–8
B.4.3 Linkage Pair B–9
B.5 Types of Routines B–10
B.5.1 Routine Capabilities B–11
B.5.2 Entry Prologue and Exit Epilogue Sequences B–11
B.6 Establishing Self-Addressability B–12
B.7 Optimization and Automatic Alignment B–13
B.7.1 Automatic Data Alignment B–13
B.7.1.1 Controlling Data Alignment B–14
B.7.1.2 Directives for Automatic Data Alignment B–14
B.7.2 Automatic Code Label Alignment B–15
B.7.3 Scheduling Optimization B–15
B.7.4 Peephole Optimization B–16
B.7.5 Using MACRO–64 for Performance B–16
B.7.6 Viewing Optimization Results B–17

C Using LSE with MACRO–64

C.1 Invoking LSE C–1
C.2 Running Diagnostics C–1

D Differences from VAX MACRO

D.1 Assembler Features in MACRO–64 . . . . . D–1
D.2 VAX MACRO Features Not Present in MACRO–64 . . . . . D–4

E Error Messages

F MACRO–64 Listing Format

Index

Examples

3–1	Lexical Processing Without the Escape Operator	3–4
3–2	Lexical Processing with Escape Operator	3–4
3–3	Using Lexical Operators	3–5
3–4	Source Statements After Macro Expansion	3–6
6–1	Program Using Supplied Macros	6–5
6–2	Program Using $CALL	6–7
B–1	Routine Call Example	B–3
B–2	Routine Without Linkage Pairs	B–9
B–3	Routine With Linkage Pairs	B–10
B–4	Establishing a Base Address	B–12
B–5	Enabling and Disabling Data Alignment	B–14
F–1	Main Source File	F–1
F–2	Include Source File	F–2
F–3	Example Listing Output	F–2

Figures

A–1	Data Structures	A–2
A–2	Instruction Formats	A–3
B–1	Base-Register Memory Displacement	B–4
B–2	Accessing Memory Through Base Addresses	B–5
B–3	Program Call Chain	B–7

Tables

2–1	Using Tab Stops in Statement Fields	2–2
2–2	Special Characters Used in MACRO–64 Statements	2–5
2–3	Summary of Unary Operators	2–17
2–4	Summary of Binary Operators	2–18
3–1	Summary of MACRO–64 Lexicals	3–7
3–2	%TYPE Attributes	3–19
5–1	Summary of General Assembler Directives	5–3
5–2	Summary of Macro Directives	5–5
5–3	.ENABLE and .DISABLE Symbolic Arguments	5–28
5–4	Condition Tests for Conditional Assembly Directives	5–46
5–5	Program Section Attributes	5–84
5–6	Default Program Section Attributes	5–86
5–7	.SHOW and .NOSHOW Symbolic Arguments	5–94
6–1	ARGS Arguments	6–11

A–1	Register Usage Conventions for OpenVMS AXP	A–4
A–2	Instruction Operand Notation	A–4
A–3	Instruction Qualifier Notation	A–5
A–4	F-P Control Register (FPCR) Format	A–5
A–5	Decodable Pseudo-Operations	A–6
A–6	Common Architecture Opcodes in Numerical Order	A–7
A–7	OpenVMS Unprivileged PALcode Instructions	A–11
A–8	OpenVMS Privileged PALcode Instructions	A–12
A–9	PALcode Opcodes in Numerical Order	A–13
A–10	Common Architecture Instructions	A–15
B–1	Frame Attributes	B–11
B–2	Directives Using Automatic Alignment	B–14

Preface

This manual describes the MACRO–64 assembly language.

Intended Audience

This manual is intended for all programmers writing MACRO–64 programs. Before reading this manual, you should be familiar with assembly language programming, the Alpha AXP architecture and instruction set, and the operating system in use on your AXP system.

Document Structure

This manual contains the following chapters and appendices:

- Chapter 1 introduces the features of the MACRO–64 language.
- Chapter 2 describes the format of the MACRO–64 source statements.
- Chapter 3 describes the MACRO–64 lexical operators. It also discusses differences and similarities between the VAX MACRO string operators and those of MACRO–64.
- Chapter 4 describes the arguments used with macros.
- Chapter 5 describes the MACRO–64 general assembler directives and the directives used in defining and expanding macros.
- Chapter 6 describes the macros provided with the MACRO–64 assembler.
- Appendix A provides a quick reference to the Alpha AXP architecture as it relates to the MACRO–64 programming language. For more information, see the *Alpha Architecture Handbook*.
- Appendix B provides information on how to program more effectively using the MACRO–64 assembler.
- Appendix C explains how to invoke the DEC Language-Sensitive Editor (LSE) with the MACRO–64 language.
- Appendix D describes the differences between VAX MACRO and MACRO–64 features.
- Appendix E describes the error messages produced by the MACRO–64 assembler.
- Appendix F provides a MACRO–64 binary listing format example.

Associated Documents

For more information on MACRO–64, see the following guide in this documentation set:

- *MACRO–64 Assembler for OpenVMS AXP Systems Installation Guide*

For more information on the Alpha AXP architecture, see the following:

- *Alpha Architecture Reference Manual*
- *Alpha Architecture Handbook*
- *Alpha AXP Achitecture Quick Reference Guide*
- *OpenVMS Calling Standard*

Conventions

In this manual, every use of Alpha VMS means the OpenVMS AXP operating system, every use of VAX VMS means the OpenVMS VAX operating system, and every use of VMS means both the OpenVMS AXP operating system and the OpenVMS VAX operating system.

The following conventions are used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in a VMS file specification, or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.

boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*), command lines (for example, /PRODUCER=*name*), and command parameters in text.
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal, unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)
PB1, PB2, PB3, PB4	PB1, PB2, PB3, and PB4 indicate buttons on the puck.
SB, SB	SB and SB indicate buttons on the stylus.
MACRO–64	MACRO–64 refers to MACRO–64 Assembler for OpenVMS AXP Systems.

1 Introduction

MACRO–64 Assembler for OpenVMS AXP Systems is an assembly language for programming Alpha AXP computers. Source programs written in MACRO–64 are translated into object (or binary) code by the MACRO–64 assembler, which produces an object module and, optionally, a listing file. The features of the language are introduced in this chapter, as well as instructions to invoke the assembler.

1.1 Language Features

MACRO–64 source programs consist of a sequence of source statements. These source statements may be any of the following:

- Alpha AXP native-mode instructions
- Direct assignment statements
- Assembler directives

1.1.1 Alpha AXP Native-Mode Instructions

Instructions manipulate data. They perform such functions as addition, data conversion, and transfer of control. Instructions are usually followed in the source statement by operands, which can be any kind of data needed for the operation of the instruction. For a brief description of the Alpha instruction set, see Appendix A. For a more detailed description of the Alpha native-mode instruction set, see the *Alpha Architecture Reference Manual* or the *Alpha Architecture Handbook*.

1.1.2 Direct Assignment Statements

Direct assignment statements equate symbols to values. For more information on direct assignment statements, see Section 2.11.

1.1.3 Assembler Directives

Assembler directives guide the assembly process and provide tools for using the instructions. For more information on assembler directives, see Chapter 5.

There are two classes of assembler directives: general assembler directives and macro directives.

1.1.3.1 General Assembler Directives

You can use general assembler directives to perform the following operations:

- Store data or reserve memory for data storage
- Control the alignment of parts of the program in memory
- Specify the methods of accessing the sections of memory in which the program will be stored

- Specify a procedure in the program
- Specify the way in which symbols will be referenced
- Specify that a part of the program is to be assembled only under certain conditions
- Display informational and diagnostic messages
- Control the assembler options that are used to interpret the source program

1.1.3.2 Macro Directives

Macro directives are used to define macros and repeat blocks. They allow you to repeat identical or similar sequences of source statements throughout a program without rewriting those sequences. Use of macros and repeat blocks helps minimize programmer errors and speeds the debugging process.

1.2 Invoking MACRO–64 on OpenVMS Systems

To invoke MACRO–64, enter the MACRO command and the /Alpha_AXP command-line qualifier, using the following syntax:

```
MACRO/Alpha_AXP  file-spec[, . . . ]
```

You must specify the /Alpha_AXP command-line qualifier before any other command-line parameters, qualifiers, or file specifications.

file-spec[, . . .]

If you do not specify a file type for an input file, the assembler uses the default file type of .M64.

You can specify one or more source files to be assembled. To assemble files individually, separate the file specifications with commas. To concatenate and assemble the files as a single input file, separate the file specifications with plus signs (+).

Command-line qualifiers control special assembler options. Assembler options can apply to the entire MACRO/Alpha_AXP command-line, or to the individual file being assembled. When the qualifier follows the MACRO/Alpha_AXP command, it applies to all files listed. For more information on command-line qualifiers, see Section 1.2.1.

1.2.1 Command-Line Qualifiers

This section describes the command qualifiers used with the MACRO/Alpha_AXP command.

/[NO]ALIGNMENT=option

Controls the alignment of code and data. Valid options are:

Option	Function
CODE	Alignment of certain branch target labels.
DATA	Natural alignment of data items.

If you omit the qualifier from the command line, the default options are /NOALIGNMENT=(CODE, DATA). If more than one option is specified, the options must be enclosed in parentheses and separated by a comma.

/[NO]DEBUG[=(options)]

Specifies DEBUG support. Valid options are:

Option	Function
SYMBOL	Generates debug symbol information.
TRACEBACK	Generates traceback information.
ALL	Generates all previous debug information.
NONE	Generates no debug information.

The default qualifier is /NODEBUG. When you specify /DEBUG with no options, the default option is /DEBUG=ALL.

/[NO]DEFINE=(symbol [=[=]value],...)

Performs the same function as direct symbol assignment in your source program. That is, the /DEFINE qualifier defines a numeric symbol. For additional information, see Section 2.11.

The Digital Command Language (DCL) converts all input to uppercase unless you enclose it within quotation marks.

Use a single equal sign between the symbol and the value to define a local symbol. Use two equal signs between the symbol and the value to define a global symbol. The final value of a global symbol is output to the object module and is available during the linking process. A local symbol is only available during the assembly process.

You cannot define a lexical string symbol with /DEFINE. The value you specify for a symbol must be an integer literal. You can specify this value using a binary, octal, decimal, or hexadecimal radix. The default radix is decimal. To specify an alternate radix, use the format described in Section 2.9.1. If you specify an alternate radix, you must use MACRO–64 radix syntax, not DCL radix syntax.

If you do not specify a value for the symbol, it defaults to 1.

The simplest form of a /DEFINE definition is as follows:

```
/DEFINE=TRUE
```

This definition is equivalent to the following definition:

```
TRUE=1
```

You can also specify more than one symbol definition as with the following command:

```
/DEFINE=(CHIP==21064,UNROLL=4)
```

This definition is equivalent to the following definitions:

```
CHIP==21064
UNROLL=4
```

When more than one /DEFINE qualifier is present on the MACRO command line or in a single assembly unit, the assembler uses only the last one.

The default qualifier is /NODEFINE.

/[NO]DIAGNOSTIC[=file-spec]

Controls whether diagnostics are created and stored in the specified optional file. If a file specification is not supplied, the assembler creates a diagnostic file using the same name as the source file. For example, if you use a source file named XXX.M64, the assembler creates a diagnostic file named XXX.DIA. You can use the diagnostic file with other Digital layered products including, but not limited to, the DEC Language-Sensitive Editor (LSE).

The default qualifier is /NODIAGNOSTIC.

/ENVIRONMENT=[NO]FLOAT

Controls whether the assembler generates floating-point instructions when optimizing code and performing code-label alignment.

Currently, the only floating-point instruction generated by the assembler during optimization and alignment processing is FNOP, the floating-point no-operation instruction. If you specify /ENVIRONMENT=NOFLOAT, the assembler does not generate any floating-point instructions as part of optimization and alignment processing. Floating-point instructions that you specify in your source program are unaffected.

/LIBRARY

Searches macro libraries in the following order:

1. The library designated by the /LIBRARY qualifier.
2. The .LIBRARY directives.
3. The MACRO64.MLB library. The assembler searches for the MACRO64.MLB macro library in the following locations: MACRO64$LIBRARY, ALPHA$LIBRARY, and finally SYS$LIBRARY.
4. The STARLET.MLB library. The assembler searches for the STARLET.MLB macro library in the following locations: MACRO64$LIBRARY, ALPHA$LIBRARY, and finally SYS$LIBRARY.

In addition, you can place the macro library definitions in the listing file by using the command-line qualifier /SHOW=LIBRARY.

/[NO]LIST[=file-spec]

Controls whether a listing is created and optionally provides an output file specification for the listing file. Do not use wildcard characters in this file specification.

If you enter the MACRO/Alpha_AXP command interactively, the default qualifier is /NOLIST. The assembler sends output to the current output device rather than to a listing file. If you execute the MACRO/Alpha_AXP command in a batch job, the default qualifier is /LIST.

If you do not specify a file specification, the assembler creates a listing file using the same name as the source file. For example, if you use a source file named XXX.M64, the assembler creates a listing file named XXX.LIS.

/[NO]MACHINE_CODE

Produces a binary machine code listing after the source text if a listing file is requested. The default qualifier is /NOMACHINE_CODE.

/NAMES=case_option

Specifies the alphabetic casing of identifiers in source code statements. Valid options are:

Option	Function
UPPER_ CASE	Converts all identifiers to upper alphabetic case.
LOWER_ CASE	Converts all identifiers to lower alphabetic case.
AS_IS	Causes all identifiers to remain in the case used in source statements.

If you use the /NAMES qualifier in a command line, you must supply a case_ option. If you omit the qualifier from the command line, the default option is /NAMES=UPPER_CASE.

/[NO]OBJECT[=file-spec]

Controls whether an object file is created and optionally provides a file specification. Do not use wildcard characters in this file specification.

If you do not specify a file specification, the assembler creates an object file using the same name as the source file. For example, if you use a source file named XXX.M64, the assembler creates an object file named XXX.OBJ.

The default qualifier is /OBJECT.

/[NO]OPTIMIZE[=(option-list)]

Specifies optional assembler optimizations. Valid items in the option-list are:

Option	Function
SCHEDULE	Specifies instruction scheduling.
PEEPHOLE	Specifies peepholing.

Specifying /OPTIMIZE with no options is the same as specifying /OPTIMIZE=(PEEPHOLE,SCHEDULE).

The default qualifier is /NOOPTIMIZE. See Section B.7 for information on optimizations.

/PREPROCESSOR_ONLY [=filespec]

Causes the assembler to output a source file that is the result of the input source file after preprocessing. Suppresses diagnostic messages and does not produce diagnostic (.ANA) or object (.OBJ) files.

The default option, /NOPREPROCESSOR_ONLY–MACRO–64, assembles your source files normally.

If you specify /PREPROCESSOR_ONLY without a file specification argument, the output file name defaults to the name of the primary source input file. The output file type defaults to .ASM.

The following MACRO–64 directives are executed by the preprocessor and screened from the preprocessor output file:

```
.DISABLE PREPROCESSOR_OUTPUT
.ENABLE PREPROCESSOR_OUTPUT
.IF
.ELSE
.ENDC
.IIF
.IF_FALSE
.IF_TRUE
.IF_TRUE_FALSE
.INCLUDE
.LIBRARY
.IRP
.IRPC
.REPEAT
.ENDR
.MACRO
.ENDM
.MCALL
.MDELETE
.MEXIT
.NARG
.NCHR
```

Include files are inserted in place of the .INCLUDE directive into the preprocessor output file.

Macro definitions and repeat block definitions are screened from the preprocessor output file.

Macro expansion lines and repeat block expansion lines are inserted in place of the macro invocation line or repeat block, respectively, into the preprocessor output file.

Lexical string-symbol assignment statements are screened from the preprocessor output file.

Lines containing lexical operators are replaced with their equivalents after lexical operator processing.

Lexical line continuations are processed into a single, uncontinued line.

All other language elements, including directives not previously listed, label definitions, direct numeric-symbol assignments, and so forth are passed through unchanged to the preprocessor output file.

For more information, see the description of the PREPROCESSOR_OUTPUT option with the .ENABLE directive in Chapter 5.

/[NO]SHOW=(item,...)

Modifies the output listing file. This qualifier is meaningful only when /LIST is specified. Valid items are:

Option	Function
BINARY	Lists macro expansions that generate binary code. BINARY is a subset of EXPANSIONS.
CONDITIONALS	Shows sections of code conditionally skipped.
EXPANSIONS	Shows macro expansions.
INCLUDE	Shows all .INCLUDE files.
LIBRARY	Shows macro library modules.

The default option is /SHOW=CONDITIONALS.

/[NO]WARNINGS=(option-list)

Controls the severity level of messages and diagnostics. Valid options are:

Option	Function
WARNINGS	Display/suppress warnings.
INFORMATIONALS	Display/suppress informationals.
ALL	Display/suppress warnings and informationals.
NONE	Display/suppress nothing.

The default options are /WARNINGS=(WARNINGS,INFORMATIONALS). If more than one option is specified, options must be enclosed in parentheses separated by a comma.

2

Components of MACRO–64 Source Statements

A source program consists of a sequence of source statements that the assembler interprets and processes, one at a time, generating object code or performing a specific assembly-time process. A source statement can occupy one source line or can extend onto several source lines.

This chapter describes the format of the source statement and the following components:

- Character set (Section 2.2)
- Numbers (Section 2.3)
- Quoted literals (Section 2.4)
- Symbols (Section 2.5)
- Temporary labels (Section 2.6)
- Label addresses (Section 2.7)
- Terms and expressions (Section 2.8)
- Unary operators (Section 2.9)
- Binary operators (Section 2.10)
- Direct assignment statements (Section 2.11)
- Current location counter (Section 2.12)

2.1 Source Statement Format

MACRO–64 source statements can have a maximum of four fields, as follows:

- Label field—Symbolically defines a location in a program.
- Operator field—Specifies the action to be performed by the statement; this can be an instruction, an assembler directive, or a macro call.
- Operand field—Contains the instruction operands or the assembler directive arguments or the macro arguments.
- Comment field—Contains a comment that explains the meaning of the statement; this does not affect program execution.

You can separate statement fields by either a space or a tab stop, but Digital recommends that you format statements with the Tab key to ensure consistency and clarity. See Table 2–1.

Table 2–1 Using Tab Stops in Statement Fields

Field	Column in Which Field Begins	Tab Stops to Reach Column
Label	1	0
Operator	9	1
Operand	17	2
Comment	41	5

The following example shows a typical source statement:

```
EXP:    .BLKL   50          ; Table stores expected values
```

Rules for Coding Source Statements

The following rules apply for coding source statements:

- You can continue a single statement on several lines by using a hyphen (-) as the last nonblank character before the comment field, or at the end of line (when there is no comment).
- In most cases, you can continue a statement at any point. If a symbol name is continued and the first character on the second line is a tab or a blank, the symbol name is terminated at that character. For information on using symbols, see Section 2.5.
- Blank lines are legal, but they have no significance in the source program except that they terminate a continued line.

The following sections describe each of the statement fields in detail.

2.1.1 Label Field

A label is a user-defined symbol that identifies a location in the program. The symbol is assigned a value equal to the location counter where the label occurs. The following rules apply when coding source statements with labels:

- If a statement contains a label, the label must be in the first field on the line.
- The user-defined symbol name can be up to 31 characters long and can contain any alphanumeric character, as well as the underscore (_), dollar sign ($), and period (.) characters.
- If a label extends past column 7, Digital recommends you place it on a line by itself so that the following operator field can start in column 9 of the next line.
- A label is terminated by a colon (:) or a double colon (::).

In the following source statement, EXP: is the label field:

```
EXP:    .BLKL   50          ; Table stores expected values
```

See Section 2.5.3 for a description of the rules for forming user-defined symbol names.

There are three types of labels:

- Global labels—Can be referenced by other object modules and are defined using a double colon (::).

- Local labels—Can be referenced only within the current module and are defined using a single colon (:).
- Temporary labels—Can be referenced only within the bounds of either two local labels, two global labels, two psects (program sections), or within the bounds of a temporary label scope, as defined by .ENABLE LOCAL_BLOCK and .DISABLE LOCAL_BLOCK. Temporary labels are defined using one to five digits followed by a dollar sign and a single colon ($:). See Section 2.6 for further information.

The following example shows how these labels appear in use:

```
EXP:    .BLKL   50          ; EXP is a local label used to
                            ; identify a 50-word block of storage
DATA::  .BLKW   25          ; DATA is a global label used to
                            ; identify a 25-word block of storage
10$:    .BLKW    5          ; 10$ is a temporary label used to
                            ; identify a five word block of
                            ; storage.
```

2.1.2 Operator Field

The operator field specifies the action to be performed by the statement. This field can contain an instruction, an assembler directive, or a macro call. If the operator is:

- An instruction, MACRO–64 generates the binary code for that instruction in the object module. The instructions are listed in Appendix A.
- A directive, MACRO–64 performs certain control actions or processing operations during source program assembly. The assembler directives are described in Chapter 5.
- A macro call, MACRO–64 expands the macro. Macro calls are described in Chapter 4 and in Chapter 5 (.MACRO directive). Macros supplied with MACRO–64 are described in Chapter 6.

Use either a space or a tab stop to terminate the operator field; however, Digital recommends that you use the tab stop to terminate the operator field.

In the following source statement, .BLKL is the operator field:

```
EXP:    .BLKL   50          ; Table stores expected values
```

2.1.3 Operand Field

The operand field can contain operands for instructions or arguments for either assembler directives or macro calls.

Operands for instructions identify the memory locations or the registers that are used by the machine operation. The operand field for a specific instruction must contain the correct number and type of operands required by that instruction.

Arguments for a directive must meet the format requirements of that directive. Chapter 5 describes the directives and the format of their arguments.

Operands for a macro must meet the requirements specified in the macro definition. See the description of the .MACRO directive in Chapter 5.

Use a comma (,) to separate operands for instructions and directives. (See Section 2.8 for a discussion of expressions.)

The semicolon that starts the comment field terminates the operand field. If a line does not have a comment field, the operand field is terminated by the end of the line.

In the following source statement example, 50 is the operand field supplied to the operator field, .BLKL:

```
EXP:    .BLKL   50          ; Table stores expected values
```

2.1.4 Comment Field

The comment field contains text that explains the function of the statement. You can use comments to describe algorithms, reasons for selecting particular methods, and parameters passed to routines. Comments do not affect assembly processing or program execution.

The comment field must be preceded by a semicolon (;) and can contain any printable ASCII character. It is terminated by the end of the line.

A comment can appear on a line by itself. If a comment does not fit on one line, it can be continued on the next, but the continuation must be preceded by another semicolon.

In the following source statement example, "Table stores expected values" is the comment field. Note that the comment field begins with a semicolon.

```
EXP:    .BLKL   50          ; Table stores expected values
```

2.2 Character Set

When coding source statements, you need to be aware of what characters are acceptable to the assembler, and how the assembler interprets them. The following numbers and characters are accepted by the assembler:

- The letters of the alphabet, A to Z, uppercase and lowercase. By default, the assembler converts all lowercase letters to uppercase. This means it considers lowercase letters equivalent to uppercase letters.

  The assembler can operate in a case-sensitive mode. In case-sensitive mode, the assembler does not convert lowercase letters to uppercase letters. On OpenVMS and OpenVMS AXP systems, you can select case-sensitive mode from the command line with the /NAMES=AS_IS qualifier.

- The digits 0 to 9.

- The special characters listed in Table 2–2.

Table 2–2 Special Characters Used in MACRO–64 Statements

Character	Character Name	Function
_	Underscore	Character in symbol names.
$	Dollar sign	Character in symbol names.
.	Period	Character in symbol names, current location counter, and decimal point.
:	Colon	Local label terminator.
::	Double colon	Global label terminator.
=	Equal sign	Local direct assignment operator and macro keyword argument terminator.
==	Double equal sign	Global direct assignment operator.
#	Number sign	Literal value indicator.
@	At sign	Arithmetic shift operator.
;	Semicolon	Comment field indicator.
+	Plus sign	Unary plus operator and arithmetic addition operator.
–	Minus sign or hyphen	Unary minus operator, arithmetic subtraction operator, and line continuation indicator.
*	Asterisk	Arithmetic multiplication operator.
/	Slash	Arithmetic division operator.
&	Ampersand	Logical AND operator.
!	Exclamation point	Logical inclusive OR operator.
\	Backslash	Logical exclusive OR and numeric conversion indicator in macro arguments.
^	Circumflex	Unary operators and macro argument delimiter.
( )	Parentheses	Displacement and register field delimiter in an instruction operand. Argument delimiter to a lexical operator.
<>	Angle brackets	Argument or expression grouping delimiters.
?	Question mark	Created local label indicator in macro arguments.
’	Apostrophe	Macro argument concatenation indicator.
"	Double quote	Quoted literal delimiter.
%	Percent sign	Delimits the beginning of a lexical operator.
(space) *(tab)*	Space or tab	Separates source statement fields. Spaces within expressions are otherwise ignored.
,	Comma	Separates symbolic arguments within the operand field. Multiple expressions in the operand field must be separated by commas.

2.3 Numbers

Numbers can be integers or floating-point numbers. The following sections describe these types of numbers.

2.3.1 Integers

You can use integers in any expression, including expressions in operands and in direct assignment statements. For information on expressions, see Section 2.8.

FORMAT

snn

s
An optional sign: plus sign (+) for positive numbers (the default), or minus sign (−) for negative numbers.

nn
A string of numeric characters that is legal for the specified radix.

MACRO–64 interprets all integers in the source program as decimal unless the number is preceded by a radix control operator. For more information on radix control operators, see Section 2.9.1.

Integers must be in the range of -2^{63} to $+2^{63}-1$ for signed data or in the range of 0 to $2^{64}-1$ for unsigned data.

Negative numbers must be preceded by a minus sign; MACRO–64 translates such numbers into two's complement form. In positive numbers, the plus sign is optional.

2.3.2 Floating-Point Numbers

You can use a floating-point number in the .DOUBLE, .FLOAT, .F_FLOATING, .D_FLOATING, .G_FLOATING, .S_FLOATING, and .T_FLOATING directives. For more information on directives, see Chapter 5. You cannot use a floating-point number in an expression or with a unary or binary operator except the unary plus and unary minus. For information on unary and binary operators, see Section 2.9 and Section 2.10.

You can specify a floating-point number with or without an exponent.

FORMAT

Floating-point number without exponent:

snn
snn.nn
snn.

Floating-point number with exponent (E):

snnEsnn
snn.nnEsnn
snn.Esnn

s
An optional sign.

nn
A string of decimal digits in the range of 0 to 9.

The decimal point can appear anywhere to the right of the first digit. A floating-point number cannot start with a decimal point because MACRO–64 treats the number as a user-defined symbol. For information on user-defined symbols, see Section 2.5.3.

2.4 Quoted Literals

A quoted literal is a string of characters enclosed in double quotes (" "). Use the following guidelines when specifying characters in a quoted literal:

- Any character except null, carriage return, and form feed can appear within the string.
- To include a double quote or backslash in a string, you must precede it with a backslash (\ ).
- To specify an arbitrary character, you can specify "*xhh*", where each *h* represents a single hexadecimal digit. For example:

  ```
  "AB\\CD\"EF\x47"
  ```

  This string contains the following characters:

  ```
  AB\CD"EFG
  ```

Also note that the assembler does not convert the case of alphabetic characters within a quoted literal.

Quoted literals can be continued over several lines. Use the hyphen (-) as the line continuation character and delimit the string with double quotes. For example:

```
.ASCII "Strings are delimited with double quotes."
.ASCII "The backslash is an escape character."
.ASCII "Strings can be continued onto multiple lines -
just as any other line."
.ASCII "Use two backslashes (\\) to represent -
the back-slash itself."
.ASCII "Hexidecimal escape sequences use -
lower or upper X: \x00 or \X00"
.ASCII "Precede a double quote with a backslash (\") -
to embed the quote."
```

2.5 Symbols

You use symbols in MACRO–64 source statements to represent an instruction, directive, register name, or value. You can use four types of symbols in MACRO–64 source programs: permanent symbols, predefined symbols, user-defined symbols, and macro names.

2.5.1 Permanent Symbols

Permanent symbols consist of MACRO–64 directives and instruction mnemonics. You need not define directives before you use them in the operator field of a MACRO–64 source statement. It is also not necessary to define instruction mnemonics before using them in the instruction statements.

2.5.2 Predefined Symbols

Predefined symbols are MACRO–64 register symbols that are not *permanently* reserved. You can delete the definition of any of these predefined register symbols. You can also define your own register symbols.

You can express the 32 general registers and the 32 floating-point registers of the Alpha AXP processor in a source program as follows:

Register Name	Description
R0	General register 0.
R1	General register 1.
.	.
.	.
.	.
R29 or FP	General register 29 or frame pointer. If you use R29 as a frame pointer, Digital recommends you use the name FP. If you use R29 as a general register, Digital recommends you use the name R29.
R30 or SP	General register 30 or stack pointer. If you use R30 as a stack pointer, the name SP is recommended; if you use R30 as a general register, the name R30 is recommended.
R31	General register 31.
F0	Floating-point register 0.
.	.
.	.
.	.
F31	Floating-point register 31.

Note

When MACRO–64 operates in /NAMES=AS_IS mode, all of the previous register symbols are defined in all uppercase and all lowercase.

To define your own register symbols, use either the .DEFINE_FREG or .DEFINE_IREG directive for floating-point or integer registers, respectively. For more information about the .DEFINE_FREG or .DEFINE_IREG directives, see the description in Chapter 5.

You can delete a register symbol definition with the .UNDEFINE_REG directive. For more information about the .UNDEFINE_REG directive, see the description in Chapter 5.

While an identifier is defined as a register symbol, it can only be used in those contexts that allow a register.

2.5.3 User-Defined Symbols and Macro Names

You can use symbols to define labels, or you can equate them to a specific value by a direct assignment statement. (See Section 2.11.) You can also use these symbols in expressions. For more information on expressions, see Section 2.8.

Use the following rules to create user-defined symbols:

- Use alphanumeric characters, underscores (_), dollar signs ($), and periods (.). Any other character terminates the symbol.
- The first character of a symbol cannot be a number.
- The symbol must be no more than 31 characters long and must be unique.
- The symbol must not be a register name.

- The symbol cannot be one of the following conditional or macro directives:

.ELSE	.ENDC	.ENDM
.ENDR	.IF	.IF_FALSE (.IFF)
.IF_TRUE (.IFT)	.IF_TRUE_FALSE (.IFTF)	.IIF
.INCLUDE	.IRP	.IRPC
.LIBRARY	.MACRO	.MCALL
.MDELETE	.MEXIT	.NARG
.NCHAR	.REPEAT	

In addition, by Digital convention:

- The dollar sign ($) is reserved for names defined by Digital. This convention ensures that a user-defined name (that does not have a dollar sign) will not conflict with a Digital-defined name (that does have a dollar sign).
- Do not use the period (.) in any global symbol name because many languages, such as Fortran, do not allow periods in symbol names.

Macro names follow the same rules and conventions as user-defined symbols. See the description of the .MACRO directive in Chapter 5 for more information on macro names. User-defined symbols and macro names do not conflict; that is, you can use the same name for a user-defined symbol and a macro.

2.5.4 Determining Symbol Values

The value of a symbol depends on its use in the program. MACRO–64 uses a different method to determine the values of symbols in the operator field than it uses to determine the values of symbols in the operand field.

2.5.4.1 Using Symbols in the Operator Field

A symbol in the operator field can be either a permanent symbol or a macro name. MACRO–64 searches for a symbol definition in the following order:

1. Macro and conditional directives:

.ELSE	.ENDC	.ENDM
.ENDR	.IF	.IF_FALSE (.IFF)
.IF_TRUE (.IFT)	.IF_TRUE_FALSE (.IFTF)	.IIF
.INCLUDE	.IRP	.IRPC
.LIBRARY	.MACRO	.MCALL
.MDELETE	.MEXIT	.NARG
.NCHAR	.REPEAT	

2. Previously defined macro names
3. Permanent symbols (instructions and other directives)

This search order allows most permanent symbols, except conditional directives and macro directives, to be redefined as macro names. If a symbol in the operator field is not defined as a macro or a permanent symbol, the assembler displays an error message.

2.5.4.2 Using Symbols in the Operand Field

A symbol in the operand field must be either a user-defined numeric symbol, a label, or a register name.

User-defined numeric symbols and labels can be either local (internal) symbols or global (external) symbols. Whether numeric symbols and labels are local or global depends on their use in the source program.

You can reference a local numeric symbol or label only in the module in which it is defined. If local numeric symbols or labels with the same names are defined in different modules, the symbols and labels are completely independent. The definition of a global numeric symbol or label, however, can be referenced from any module in the program.

MACRO–64 treats all user-defined numeric symbols and labels as local unless you explicitly declare them to be global by doing one of the following:

- Use the double colon (::) in defining a label. For more information on labels, see Section 2.1.1.
- Use the double equal sign (==) in a direct assignment statement. For more information on direct assignment statements, see Section 2.11.
- Use the .WEAK directive. For more information on the .WEAK directive, see Chapter 5.

You can only use user-defined lexical string symbols with the lexical string operators. For more information on lexical string operators, see Chapter 3. You can define a macro using the same name as a previously defined local numeric symbol, global numeric symbol, or a lexical string symbol. However, you cannot define a lexical string symbol and a numeric symbol using the same name.

In addition, you cannot use the same name for both a local and global numeric symbol. You cannot use the same symbol name for both a numeric symbol (local or global) and a label (local or global).

2.6 Temporary Labels Within Source Code

Use temporary labels to identify addresses within a block of source code.

Format

nnnnn$:

nnnnn

A decimal integer in the range of 1 to 65,535.

In most cases, you can use temporary labels in the same way you use other labels that you define; however, there are some differences:

- Temporary labels cannot be referenced outside the temporary label block in which they are declared.
- Temporary labels can be redeclared in another block of source code.
- Temporary labels that occur within a psect with the MIX or NOMIX attribute do not appear in the debugger symbol table; thus, they cannot be accessed by the symbolic debugger. For more information on psects, see Chapter 5.
- Temporary labels cannot be used in the .END or .PROCEDURE_DESCRIPTOR directives. For more information on the .END directive, see Chapter 5.

By convention, temporary labels are positioned like statement labels: left-justified in the source text. Although temporary labels can appear in the program in any order, by convention, the temporary labels in any block of source code should be in increasing numeric order.

Temporary labels are useful as branch addresses when you use the address only within the block. You can use temporary labels to distinguish between addresses that are referenced only in a small block of code and addresses that are referenced elsewhere in the module. A disadvantage of temporary labels is that their numeric names do not provide any indication of their purpose. Consequently, you should not use temporary labels to label sequences of statements that are logically unrelated; you should use user-defined symbols instead.

Digital recommends that users create temporary labels only in the range of 0$ to 29999$ because the assembler automatically creates temporary labels in the range of 30000$ to 65535$ for use in macros. For more information on temporary labels, see Section 4.7.

The temporary label block in which a temporary label is valid is delimited by the following statements:

- A user-defined label: global or local.
- A .PSECT directive. For more information on the .PSECT directive, see Chapter 5.

  An example showing the correct and incorrect use of temporary labels follows:

```
A:    ADDQ R1, R2, R3
      BEQ R3, 2$          ; correct use
      MULQ R2, R3, R4
2$:   ADDQ R1, R4, R5     ; definition of temporary label
B:    BNE R5, 2$          ; illegal
C:    SUBQ R2, R4, R6
```

  In this example, 2$ is a temporary label defined in the block between A: and B:. The forward reference in the second instruction is properly resolved. The line labeled B: also references 2$, but the label B has already closed the range. The temporary label 2$ can be used later in the program, but its definition must appear within the same block as the label.

- The .ENABLE and .DISABLE directives, which can extend a local label block beyond user-defined labels and .PSECT directives. For more information on the .ENABLE and .DISABLE directives, see Chapter 5.

A temporary label block is usually delimited by two user-defined labels. However, the .ENABLE LOCAL_BLOCK directive starts a local block that is terminated by one of the following:

- A second .ENABLE LOCAL_BLOCK directive
- A .DISABLE LOCAL_BLOCK directive followed by a user-defined label or a .PSECT directive

Temporary labels can be preserved with the context of the program section in which they are defined for future reference. For more information on the .SAVE_PSECT [LOCAL_BLOCK] and .RESTORE_PSECT directives, see Chapter 5.

2.7 Label Addresses

In the absence of optimization and automatic data alignment, label addresses are defined to be the psect and offset of the current location counter (see Section 2.11) at the point where the label is defined. When either optimization or automatic data alignment are enabled, the following additional considerations apply. See Section B.7 for information on optimizations and automatic alignment.

2.7.1 Label Addresses, Optimization, and Code Alignment

Optimization and code alignment can affect the addresses assigned to labels defined in psects that have the EXE and NOMIX attributes. Optimization and code alignment are disabled by default, and can be enabled with the /OPTIMIZE and /ALIGNMENT command-line qualifiers (see Section 1.2.1) and the .ENABLE directive (see Section 5.1). In general, the assembler assigns the psect and offset of the current location counter before optimization or alignment of code labels. However, the assembler adjusts references to labels in branch instructions to the address of the label after optimization and code alignment processing. The assembler does not adjust references to labels where the label reference occurs in an expression with more than one term. The following example shows this:

```
        .PSECT CODE, EXE, NOMIX
        BSR     R0, 10$             ; R0 -> 10$ (postoptimization)
10$:    LDA     R1, 20$-10$(R0)     ; R1 -> 20$ (preoptimization)
        JMP     (R1)
        [...]
20$:
```

In the previous example, the assembler adjusts the target address of the BSR instruction to be the location of 10$ after optimization and code alignment have taken place. Thus, the branch to 10$ functions as expected. However, when processing the LDA instruction, the assembler computes the offset between 20$ and 10$ before optimization and code alignment. Thus, the address of 20$ that is stored in R1 is the address prior to optimization and code alignment. Depending on the sequence of instructions between 10$ and 20$, the address before optimization and code alignment may differ from the address after optimization and code alignment and the JMP instruction may not transfer control to the expected address.

Note also that the assembler only performs postoptimization adjustment of label addresses when the label is the only term in the expression. For example:

```
        .PSECT CODE, EXE, NOMIX
        .BASE R27,LINKAGE
        LDQ     R26, ROUTINE1_ADDR
        JSR     R26, (R26)
        LDQ     R26, ROUTINE2_ADDR
        JSR     R26, (R26)
        RET     R28
        NOP
        NOP

ROUTINE1:
        RET     (R26)
ROUTINE2:
        RET     (R26)
```

```
        .PSECT LINKAGE, NOEXE
LINKAGE:
ROUTINE1_ADDR:
        .ADDRESS ROUTINE1
ROUTINE2_ADDR:
        .ADDRESS ROUTINE2+0
```

In the previous example, the assembler adjusts the address stored with the .ADDRESS ROUTINE1 directive to the address of label ROUTINE1 after optimization and code alignment. However, since the expression in the .ADDRESS ROUTINE2+0 directive is not a single term, the assembler adds the offset 0 and the address of ROUTINE2 before optimization and code alignment and stores the result. Since the address stored is the address before optimization and code alignment, the second JSR instruction may not transfer control to the address that is expected.

2.7.2 Label Addresses and Automatic Data Alignment

Automatic data alignment can affect the addresses assigned to labels in psects that have the NOEXE or MIX attributes. Automatic data alignment is enabled with the .ENABLE ALIGN_DATA directive or the /ALIGNMENT=data command-line option. For more information on automatic alignment, see Section 5.2. For more information on the .ENABLE directive, see Section 5.3.

A label that occurs in a statement with a data-storage directive is assigned the psect and offset of the storage allocated by the data-storage directive. If the data-storage directive requires automatic alignment, the address is assigned to the label after automatic alignment.

The same is true of labels that occur in statements by themselves and that are followed by a data directive in a subsequent statement. However, if a label occurs in a statement by itself and is followed by a statement that is not a data-storage directive, a macro directive, a conditional directive, or a lexical-string symbol assignment, the label is assigned the psect and offset of the current location counter before any automatic alignment.

The assembler only assigns addresses to labels after alignment under the conditions previously described and only with automatic alignment. If you place a label before a .ALIGN directive that manually aligns the current location counter, the assembler assigns the address of the label before performing the manual alignment. The following example shows the interaction of label address assignment and automatic data alignment:

```
    .ENABLE ALIGN_DATA
    .PSECT DATA, NOEXE
    .BYTE   1       ; The byte is stored in psect data at offset 0
A:  .PRINT "Not aligned" ; Any non-macro, nonconditional
                    ; statement, including this .PRINT directive
              ; prevents A from being automatically aligned
                    ; -- A is assigned offset 1
B:                  ; B is automatically aligned to offset 4
C:  .LONG   2       ; C is automatically aligned to offset 4
D:  .ALIGN 0        ; The .ALIGN global directive prevents D
                    ;  from being automatically aligned --
      ;  D is assigned offset 8
E:  .OCTA 3         ; E is automatically aligned to offset 16
```

Automatic data alignment and label-address assignment can be an important consideration when calculating the difference between two labels. For example, consider the following macro, which stores a string preceded by a word that contains the string's length:

```
        .MACRO VARYING_STRING STRING ?L1, ?L2
                .WORD L2-L1
            L1: .ASCII "STRING"
            L2:
        .ENDM VARYING_STRING
```

If an invocation of the VARYING_STRING macro is followed by a data-storage directive that requires automatic alignment, the VARYING_STRING macro will not store the correct string length. For example:

```
        .PSECT DATA, NOEXE, .OCTA
        .ENABLE ALIGN_DATA
        VARYING_STRING <Time for something sweet!> ; 25 bytes
F:      .OCTA 4
```

In this example, the intention is to make the L2 label generated by the VARYING_STRING macro be assigned the offset 27, 2 for the .WORD directive, plus 25 for the .ASCII directive. Instead, it is assigned the offset 32 along with the label F because the .OCTA directive requires automatic alignment to offset 32. Therefore, the string length is incorrectly stored as 30 rather 25. To make this macro work as desired, you must include, in the macro definition, a macro directive that is not a data-storage, macro, or conditional directive after the generated label L2. In the following example, .ALIGN 0 is a good choice as it reflects the idea that the preceding label is not aligned:

```
 .MACRO VARYING_STRING STRING ?L1, ?L2
      .WORD L2-L1
    L1: .ASCII "STRING"
    L2: .ALIGN 0
 .ENDM VARYING_STRING
```

With this change, the generated label L2 is assigned the offset 27 before the assembler performs automatic data alignment for the .OCTA directive. As a result, the VARYING_STRING macro works as desired and stores the correct string length of 25.

2.8 Terms and Expressions

A term can be any of the following:

- A number.
- A numeric symbol.
- A label.
- The current location counter. For more information on the current location counter, see Section 2.12.
- Any of the previously noted items preceded by a unary operator.

For more information on unary operators, see Section 2.9.

MACRO–64 evaluates terms as quadword (8-byte) values. The current location counter (.) has the value of the location counter at the start of the current operand.

MACRO–64 considers unary operators part of a term and, thus, performs the action indicated by a unary operator before it performs the action indicated by a binary operator.

Expressions are combinations of terms joined by binary operators and evaluated as quadword (8-byte) values. For more information on binary operators, see Section 2.10. MACRO–64 evaluates expressions from left to right with no operator precedence rules. However, you can use angle brackets (<>) to change the order of evaluation. Any part of an expression that is enclosed in angle brackets is first evaluated to a single value, which is then used in evaluating the complete expression. For example, the expressions A*B+C and A*<B+C> are different. In the first case, A and B are multiplied and then C is added to the product. In the second case, B and C are added and the sum is multiplied by A. You can also use angle brackets to apply a unary operator to an entire expression, such as −<A+B>.

Expressions fall into four categories: relocatable, absolute, external (global), and complex. You can determine the type of expression by the following rules:

- An expression is relocatable if its value is fixed relative to the start of the psect in which it appears. The current location counter is relocatable in a relocatable psect.
- An expression is absolute if its value is an assembly-time constant. An expression whose terms are all numbers is absolute. An expression that consists of a relocatable term minus another relocatable term from the same psect is absolute, because such an expression reduces to an assembly-time constant.
- An expression is external if it is not complex, and it contains one or more symbols that are not defined in the current module.
- An expression is complex if it contains a relocatable or external term or subexpression that is operated upon by an operator other than the plus sign (+) or the minus sign (−). An expression is also complex if it contains more than one term or subexpression that is relocatable or external. An exception to this rule is the difference between two relocatable subexpressions or terms where both relocatable values occur in the same psect. In this case, the expression is absolute.

Complex expressions are constrained to use the following form:

FORMAT

term operator term

term

Term is a relocatable or external subexpression.

operator

Operator is any of the MACRO–64 operators described in Section 2.10.

Neither term can itself be a complex subexpression. If you specify a complex expression that does not match the correct form, the assembler issues a diagnostic error message indicating that the expression is too complex. Note also that the assembler does not attempt to reorder expressions to make your expressions match the correct form. For example, the following expression is too complex:

```
.external E1, E2
.quad E1+5+E2+6  ; too complex
```

Because the assembler has no precedence rules, it attempts to evaluate the previous expression as <<<E1+5>+E2>+6>. Since <<E1+5>+E2> is itself a complex term, <<<E1+5>+E2>+6> does not match the previous form and is too complex. However, you can regroup the expression using angle brackets (<>) to match the required form as follows:

```
.external E1, E2
.quad <E1+5>+<E2+6>  ; legal complex expression
```

In this case, both <E1+5> and <E2+6> are simple, external terms. Since none of the terms are complex, the expression matches the correct form and the assembler accepts the complex expression.

You can use any type of expression in most MACRO–64 statements, but restrictions are placed on expressions used in the following contexts:

- .BASE directive.
- .BLK*x* storage allocation directives.
- .IF conditional assembly block directives.
- .REPEAT repeat block directives.
- Direct assignment statements.
- Lexical string operator arguments. For more information on direct assignment statements, see Section 2.11.

See Chapter 5 for descriptions of the directives listed in the preceding list.

Expressions used in these contexts can contain only symbols or labels that have been previously defined in the current module.

The .BASE directive accepts expressions that contain external symbols previously declared with the .EXTERNAL directive. The other contexts previously described cannot accept expressions that contain external symbols. Symbols defined later in the current module cannot be used in any of these contexts.

Expressions in the .IF conditional directives, .REPEAT conditional directives, and lexical string operator arguments are relocatable. However, expressions in the .BLKx directives must be absolute.

Expressions cannot contain floating-point numbers. The floating-point data-storage directives accept constant values. They do not accept floating-point expressions. For more information on the floating-point data-storage directives, see Chapter 5.

The following example shows the use of expressions:

```
A = 2*100                ; 2*100 is an absolute expression
        .BLKB   A+50     ; A+50 is an absolute expression and
                         ;   contains no undefined symbols
LAB:    .BLKW   A        ; LAB is relocatable
HALF = LAB+<A/2>         ; LAB+<A/2> is a relocatable
                         ;   expression and contains no
                         ;   undefined symbols
LAB2:   .BLKB   LAB2-LAB ; LAB2-LAB is an absolute expression
                         ;   and contains no undefined symbols
```

2.9 Unary Operators for Terms and Expressions

A unary operator modifies a term or an expression and indicates the action to be performed on that term or expression. Expressions modified by unary operators must be enclosed in angle brackets. You can use unary operators to indicate whether a term or expression is positive or negative. If unary plus or minus is not specified, the default value is plus. In addition, unary operators perform radix conversion and numeric control operations, as described in the following sections. Table 2–3 summarizes the unary operators.

Table 2–3 Summary of Unary Operators

Unary Operator	Operator Name	Example	Operation
+	Plus sign	+A	Results in the positive value of A.
−	Minus sign	−A	Results in the negative (two's complement) value of A.
\	Value of Escape	\symbol	Indicates that the value of the symbol should be used. In a string literal, indicates an escape sequence. For example: "Bob\X0A"
^A or ^a	ASCII	^A/ABCD/	Specifies an ASCII constant.
^B or ^b	Binary	^B11000111	Specifies that 11000111 is a binary number.
^D or ^d	Decimal	^D127	Specifies that 127 is a decimal number.
^O or ^o	Octal	^O34	Specifies that 34 is an octal number.
^X or ^x	Hexadecimal	^XFCF9	Specifies that FCF9 is a hexadecimal number.
^C or ^c	Complement	^C24	Produces the one's complement value of 24 (decimal).

2.9.1 Radix Control Operators

Radix control operators determine the radix of a term or expression. MACRO–64 accepts terms or expressions in four different radixes: binary, decimal, octal, and hexadecimal. The default radix is decimal.

FORMATS

^Bnn ^Dnn ^Onn ^Xnn

nn

A string of characters that is legal in the specified radix. The following are the legal characters for each radix:

Format	Radix Name	Legal Characters
^Bnn	Binary	0 and 1
^Dnn	Decimal	0 to 9
^Onn	Octal	0 to 7
^Xnn	Hexadecimal	0 to 9 and A to F

You can include radix control operators in the source program anywhere a numeric value is legal. A radix control operator affects only the term immediately following it, causing that term to be evaluated in the specified radix. For example:

```
.WORD   ^B00001101              ; Binary radix
.WORD   ^D123                   ; Decimal radix (default)
.WORD   ^O47                    ; Octal radix
```

Do not place spaces or tabs between the circumflex (^), the radix specifier (B, D, O, or X), or the numeric value.

2.9.2 Numeric Complement Operator

The complement operator (^C) produces the one's complement of the specified value.

FORMAT

^Cterm

term

Any term or expression. If an expression is specified, it must be enclosed in angle brackets.

MACRO–64 evaluates the term or expression as an 8-byte value before complementing it. For example:

```
.LONG      ^C^XFF       ; Produces FFFFFF00 (hex)
.LONG      ^C25         ; Produces complement of
                        ;   25 (dec) which is
                        ;   FFFFFFE6 (hex)
```

2.10 Binary Operators

In contrast to unary operators, binary operators specify actions to be performed on two terms or expressions. You can enclose expressions in angle brackets to specify the order of evaluation. Table 2–4 summarizes the binary operators.

Table 2–4 Summary of Binary Operators

Binary Operator	Operator Name	Example	Operation
+	Plus sign	A+B	Addition
−	Minus sign	A−B	Subtraction
*	Asterisk	A*B	Multiplication
/	Slash	A/B	Division
@	At sign	A@B	Arithmetic shift
&	Ampersand	A&B	Logical AND (product)
!	Exclamation point	A!B	Logical OR (sum)
\	Backslash	A\B	Logical XOR (difference)

All binary operators have equal priority. You can group terms or expressions for evaluation by enclosing them in angle brackets. The enclosed terms and expressions are evaluated first, and remaining operations are performed from left to right. For example:

```
.LONG      1+2*3      ; Equals 9
.LONG      1+<2*3>    ; Equals 7
```

Note that a 64-bit result is returned from all binary operations. If you use the 64-bit result in a context requiring less than 64 bits, only the lower-order bits of the result are used. If the truncation causes a loss of significance in a data-storage directive, the assembler displays an error message.

The following sections describe the arithmetic shift, logical AND, logical inclusive OR, and logical exclusive OR operators.

2.10.1 Arithmetic Shift Operator

Use the arithmetic shift operator (@) to perform left and right arithmetic shifts of arithmetic quantities. The first argument is shifted left or right by the number of bit positions that you specify in the second argument. If the second argument is positive, the first argument is shifted left and the low-order bits are set to zero. If the second argument is negative, the first argument is shifted right and the high-order bits are set to the value of the original high-order bit (the sign bit). For example:

```
        .LONG   ^B101@4              ; Yields 1010000 (binary)
        .LONG   1@2                  ; Yields 100 (binary)
A = 4
        .LONG   1@A                  ; Yields 10000 (binary)
        .LONG   ^X1234@-A            ; Yields 123(hex)
```

2.10.2 Logical AND Operator

The logical AND operator (&) takes the logical AND of two operands. For example:

```
A = ^B1010
B = ^B1100
        .LONG   A&B             ; Yields 1000 (binary)
```

2.10.3 Logical Inclusive OR Operator

The logical inclusive OR operator (!) takes the logical inclusive OR of two operands. For example:

```
A = ^B1010
B = ^B1100
        .LONG   A!B             ; Yields 1110 (binary)
```

2.10.4 Logical Exclusive OR Operator

The logical exclusive OR operator (\ ) takes the logical exclusive OR of two arguments. For example:

```
A = ^B1010
B = ^B1100
        .LONG   A\B             ; Yields 0110 (binary)
```

2.11 Direct Assignment Statements

A direct assignment statement equates a symbol to a specific value. Unlike a symbol that you use as a label, you can redefine a symbol defined with a direct assignment statement as many times as you want.

FORMATS

symbol=expression

symbol= =expression

symbol=quoted-literal

symbol

A user-defined symbol.

expression

An expression that does not contain any undefined symbols or forward references. For more information on terms and expressions, see Section 2.8. The result must be either an absolute or relocatable value, whose value can be determined at the current point in the assembly. This form defines a numeric symbol.

The format with a single equal sign (=) defines a local symbol, and the format with a double equal sign (==) defines a global symbol. For more information about local and global symbols, see Section 2.5.4.

The following three syntactic rules apply to direct assignment statements:

- An equal sign (=) or double equal sign (==) must separate the symbol from the expression that defines its value. Spaces preceding or following the direct assignment operators have no significance in the resulting value.
- Only one symbol can be defined in a single direct assignment statement.
- A direct assignment statement can be followed only by a comment field.

For best results, Digital recommends you place the symbol in a direct assignment statement in the label field. For example:

```
A == 1                  ; The symbol 'A' is globally
                        ;   equated to the value 1

B = A@5                 ; The symbol 'B' is equated
                        ;   to 1@5 or 32(dec)

C = 127*10              ; The symbol 'C' is equated
                        ;   to 1270(dec)

D = ^X100/^X10          ; The symbol 'D' is equated
                        ;   to 10(hex)
```

quoted-literal

A literal within double quotes. This form defines a lexical string symbol. You can only use lexical string symbols with lexical string operators.

For more information on lexical string operators, see Chapter 3.

2.12 Current Location Counter

The current location counter is a counter kept by an assembler to determine the address assigned to an instruction or constant that is being assembled. The symbol for the current location counter, the period (.), represents the address of the current byte. MACRO–64 sets the current location counter to 0 at the beginning of the assembly and at the beginning of each new program section.

Every MACRO–64 source statement that allocates memory in the object module increments the value of the current location counter by the number of bytes allocated. For example, the directive, .LONG 0 increments the current location counter by 4. However, with the exception of the special form described later in

this section, a direct assignment statement does not increase the current location counter because no memory is allocated.

The current location counter can be explicitly set by a special form of the direct assignment statement. The location counter can be either incremented or decremented. This method of setting the location counter is often useful when defining data structures. Data-storage areas should not be reserved by explicitly setting the location counter; use the .BLK*x* directives. For more information on the .BLK directive, see Chapter 5.

FORMAT

.=expression

expression

An expression that does not contain any undefined or external symbols. For more information on expressions, see Section 2.8.

In a relocatable psect, the expression must be relocatable; that is, the expression must be relative to an address in the current psect. It also may be relative to the current location counter. For example:

```
. = .+40        ; Moves location counter forward
```

When a psect that you defined in the current module is continued, the current location counter is set to the last value of the current location counter in that psect.

In a psect with the EXE and NOMIX attributes:

- The location counter cannot be changed.
- If optimization is enabled, the location counter represents the location before optimization.

In a psect with either the EXE or NOMIX (or both) attributes:

- The location counter can be changed.
- If you store an initial data or instruction value in memory with a data directive such as .BYTE, .WORD, .LONG, or .QUAD or with an instruction statement, you can replace that initial value with a different initial value later in the assembly by assigning the appropriate address value to the current location counter and entering another data directive or instruction statement. However, the new value must be the same size and must start at exactly the same address as the value it replaces.

For more information on optimization, see Section 2.7.1.

3
MACRO–64 Lexical Operators

This chapter describes the MACRO–64 lexical operators.

3.1 Processing with Lexical Operators

Lexical operator processing is performed on all source lines and macro expansion lines before any other assembler processing. Thus, macro invocations, assembler directives, and instructions are subject to lexical operator processing before normal assembler processing.

Lexical operators are recognized and processed within string literals. Lexical operator processing is suppressed during macro registration in order that lexical operator evaluation may occur during macro expansion. Lexical operator evaluation is also suppressed for a range of text that is conditionally excluded with one of the .IF directives. In addition, lexical operator processing is not performed within a comment.

3.2 Lexical Operator Syntax

You invoke a lexical string operator with a percent sign followed by the lexical operator name, a left parentheses, a list of arguments separated by commas, and a right parentheses. The following example shows the lexical operator syntax:

```
.print "%EDIT(<Fred>,<upcase>)"
```

Spaces are allowed between syntax elements in the general lexical operator syntax. For example, the following syntax, including spaces, is allowed:

```
.print "%EDIT ( <Fred> , <upcase> )"
```

Spaces are also allowed between the opening and closing percent signs in a lexical substitution operator. See Section 3.4 for information on the lexical substitution operator.

```
.print "% lexical_symbol_name %"
```

Spaces are not allowed between the pair of percent signs indicating a lexical escape operator. See Section 3.5 for information on the lexical escape operator.

You can specify lexical operator arguments in the same way as macro arguments:

- A numeric symbol name preceded by a backslash (\). This construct results in the decimal value of the numeric symbol, as shown in the following example:

  ```
  \N
  ```

- Any string of characters surrounded by left- and right-angle brackets, as shown in the following example:

  ```
  <Foo bar thud>
  ```

You can nest angle brackets (<>). For example:

```
<<X+7>*17>
```

- Any string of characters surrounded by a delimiter specified after a caret character (^). You cannot nest delimiters. For example:

  ```
  ^%Foo bar thud%
  ```

- Any undelimited string of characters not separated by a space, tab, form feed, comma, equal sign, semicolon, or end of line. For example:

  ```
  A+B+C
  ```

In addition to the formats allowed for a macro argument, you can also specify lexical operator arguments as follows:

- An undelimited string of characters may also contain a string of characters enclosed within left and right parentheses. The characters between the left and right parentheses may contain space, tab, or comma delimiters. For example:

  ```
  16( R27 )
  ```

- You can use a lexical operator as an argument to another lexical operator. For example:

  ```
  %EXTRACT( %LOCATE($,X), %LENGTH(X) - %LOCATE($,X) ,X)
  ```

Except for the %TYPE lexical operator, a string symbol name supplied as a lexical operator argument is replaced with the value of the string symbol.

Each lexical operator accepts a given number of arguments and each argument has a specific type. There are three different types of arguments—string, integer, and name:

- A string argument can be any arbitrary sequence of characters.
- An integer argument must be an absolute or relocatable expression that can be resolved at that point in the assembly. A relocatable expression represents a psect and an offset within that psect. If you specify a relocatable expression for an integer argument, the assembler uses only the value of the offset within the psect. The offset value is determined before optimization and code alignment, but after data alignment.
- The name argument type is used only by the %TYPE lexical operator. The %TYPE lexical operator accepts the name of a numeric symbol, string symbol, label, psect, or a permanent symbol as its argument. Unlike the other lexical operators, if a string symbol name is specified as an argument to %TYPE, the value of the string symbol is not substituted for its name. Instead, information about the name is returned.

If you omit a string argument, the default is the empty string. An empty string is a string with no characters. If you omit an integer argument or specify an illegal expression, the default value is 0. The assembler does not issue diagnostic messages for illegal expressions used as arguments to lexical operators. If you omit the name argument or specify an illegal name to the %TYPE lexical operator, %TYPE returns a 0 value.

3.3 Numeric Symbols and Lexical String Symbols

Lexical string symbols are similar in concept and syntax to numeric symbols. MACRO–64 supports numeric symbols using the following syntax:

numeric_symbol_name = numeric_expression

MACRO–64 supports lexical string symbols using the following syntax:

string_symbol_name = "any string of characters"

The assembler differentiates between numeric symbol assignment and lexical string symbol assignment as follows:

- In a lexical string symbol assignment, a quoted string literal must appear after the equal sign.
- A lexical string symbol value is specified by the quoted string literal.

The quotes are not included in the value of the lexical string symbol. You cannot use the same name for a lexical string symbol, numeric symbol, or label.

Like numeric symbols, lexical string symbols are assembly time variables. After you assign a string value to a lexical string symbol, you can reassign a different value to that symbol later in the assembly.

You can use lexical string symbols as arguments to lexical operators. In particular, you can use a lexical string symbol as an argument to the lexical substitution operator (%) or the %STRING lexical operator to substitute the value of the lexical string symbol at any point in the text of your program.

3.4 Lexical Substitution Operator

You can use the lexical substitution operator at any point in your program to cause the assembler to substitute the value of a lexical string symbol for the name of the symbol. The lexical substitution operator is the percent sign (%). Place the lexical substitution operator to the left and right of the name of the lexical string symbol that you wish to subsitute, as follows:

```
%lexsym_name%
```

For example:

```
HORSES    = "All the king's horses"
MEN       = "all the king's men"
.print "%HORSES% and %MEN%"
```

This example defines two lexical string symbols: HORSES and MEN. The third statement displays a message at assembly time. The text of the message specifies that the value of the HORSES and MEN lexical string symbols be substituted as indicated. After lexical processing, the third statement appears as:

```
.print "All the king's horses and all the king's men"
```

3.5 Lexical Escape Operator

It is possible to defer the processing of a lexical string operator by using the lexical escape operator, which is the percent sign (%). Since all lexical string operators begin with a percent sign, the effect of placing two percent signs before the name of the lexical string operator defers the evaluation of the lexical string operator. If you want to defer processing of a lexical substitution operator, place

two percent signs to the left and two percent signs to the right of the lexical string symbol name.

This can be useful when you want the evaluation of a lexical string operator that you have used in a default macro argument to occur during macro expansion, rather than during macro definition. Lexical operator processing is suppressed during macro registration. Therefore, lexical operator processing is automatically deferred within the body of a macro. However, the .MACRO directive line that begins the macro definition is subject to normal lexical operator processing. Sometimes you may need to use the value of a lexical string symbol as the default for a macro argument, but you need to use the value of the lexical string symbol that is current when the macro expands, not when the macro is defined. Example 3–1 shows an example of this, but it does not use an escape operator.

Example 3–1 Lexical Processing Without the Escape Operator

```
CODE_PSECT_NAME = "CODE1"
.MACRO CODE_PSECT PSECT_NAME=%string(CODE_PSECT_NAME)
 .PSECT PSECT_NAME
.ENDM CODE_PSECT
CODE_PSECT
CODE_PSECT_NAME = "CODE2"
CODE_PSECT
```

Example 3–1 does not process correctly for the following reasons:

- The lexical operator in the .MACRO directive line is processed when the macro is defined, not when the macro expands.
- The CODE_PSECT macro always defaults to setting the psect to the CODE1 psect because the default for the PSECT_NAME argument will be set to CODE1, not %string(CODE_PSECT_NAME). This is because %string(CODE_PSECT_NAME) is evaluated when the CODE_PSECT macro is defined, not when it expands.

Example 3–2 is similar to Example 3–1 except it uses the lexical escape operator.

Example 3–2 Lexical Processing with Escape Operator

```
CODE_PSECT_NAME = "CODE1"
.macro CODE_PSECT PSECT_NAME=%%string(CODE_PSECT_NAME)
.psect PSECT_NAME
.endm CODE_PSECT
CODE_PSECT
CODE_PSECT_NAME = "CODE2"
CODE_PSECT
```

Example 3–2 processes correctly for the following reasons:

- Lexical operator processing of %%string(CODE_PSECT_NAME) is deferred when the CODE_PSECT macro is defined. The default value for the PSECT_NAME argument is stored as %string(CODE_PSECT_NAME).
- During macro expansion, %string(CODE_PSECT_NAME) is evaluated, which results in the current value of the CODE_PSECT_NAME lexical string symbol as desired.

3.6 Using Lexical Operators

This section contains two programming examples. Example 3–3 shows source statements using lexical operators. Example 3–4 shows the same source statements after macro expansion and lexical operator processing.

Example 3–3 Using Lexical Operators

```
        ; Macro to load a general or floating register
        .macro LOAD REG LOCATION

            ; If the location doesn't specify a base register,
            ; load relative to the procedure descriptor PD off
            ; of R27.
            .if lt, %locate(<(>, LOCATION), %length(LOCATION)
                ; base register specified
                LOC = "LOCATION"
            .else
                ; base register not specified
                LOC = "LOCATION-PD(R27)"
            .endc

            ; Use LDQ if a general register is specified
            .if ne, <%type(REG) & MACRO64$TYPE_GENREG>
                LDQ   REG,%string(LOC)

            .else
                ; Use LDx if a floating register is specified
                .if ne <%type(REG) & MACRO64$TYPE_FLTREG>
                    ; Use the appropriate LDx floating load
                    ; depending on the first letter of the
                    ; location to load.
                    FTYPES = "FGST"
                    .if ge, %locate(%extract(0,1,LOC),FTYPES), -
                        %length(FTYPES)
                        .error -
         "Unknown floating data type: %extract(0,1,LOC)"
                        .mexit
                    .endc
                    LD%extract(0,1,LOCATION) REG,%string(LOC)
                .else
                    .error -
         "First argument must be a general or floating register"
                .endc
            .endc
        .endm LOAD
        .psect D,noexe
pd:     .procedure_descriptor my_routine,my_entry
        ; Fill in procedure descriptor details...
BUILD_TIME:
        .asciz       "%time()"
K:      .quad        %repeat(9,<42,>)42
GPI:    .g_floating  3.14159
CIRCLE: .address     FAC_CIRCLE
TX      = 0
TY      = 8
TRADIUS = 16
        .psect X,exe
MY_ENTRY:
        LOAD  R1,K
        LOAD  R2,CIRCLE
        LOAD  F13,TRADIUS(R2)
```

(continued on next page)

Example 3–3 (Cont.) Using Lexical Operators

```
        LOAD  F11,GPI
        LOAD  F10,K
        LOAD  K
```

Example 3–4 Source Statements After Macro Expansion

```
        .psect D,noexe
pd:     .procedure_descriptor my_routine,my_entry
        ; Fill in procedure descriptor details...
BUILD_TIME:
        .asciz       " 9-JUL-1992 17:56:20"
K:      .quad        42,42,42,42,42,42,42,42,42,42
GPI:    .g_floating  3.14159
CIRCLE: .address     FAC_CIRCLE
TX      = 0
TY      = 8
TRADIUS = 16
        .psect X,exe
MY_ENTRY:
        LDQ   R1,K-PD(R27)
        LDQ   R2,CIRCLE-PD(R27)
        LDT   F13,TRADIUS(R2)
        LDG F11,GPI-PD(R27)
        .error "Unknown floating data type: K"
        .error "First argument must be a general or floating register"
```

3.7 Lexical Operators

This section describes the MACRO–64 lexical operators, their return values, and their arguments. Table 3–1 lists the lexicals and gives a brief summary of their functions. A decimal return value results in the decimal digits of the numeric value being substituted for the lexical operator.

Table 3–1 Summary of MACRO–64 Lexicals

Lexical	Return Type	Function
%EDIT	String	Lexical operator for editing text strings.
%ELEMENT	String	Lexical operator for extracting an element from a list of elements.
%EXTRACT	String	Lexical operator for extracting a range of characters from a string of characters.
%FREG	String	Lexical operator for obtaining the floating-point register number associated with a symbol.
%INTEGER	Decimal	Lexical operator to convert the value of an expression to a decimal value.
%IREG	String	Lexical operator for obtaining the integer register number associated with a symbol.
%LENGTH	Decimal	Lexical operator for determining the length of a string.
%LOCATE	Decimal	Lexical operator for locating a string of text within a another string of text.
%REPEAT	String	Lexical operator for repeating a specified string *x* number of times.
%STRING	String	Lexical operator for obtaining the value of a lexical string symbol.
%TIME	String	Lexical operator for obtaining the date and time of the assembly unit.
%TYPE	Decimal	Lexical operator for obtaining information about a name.

%EDIT

Lexical operator for editing text strings.

Format

%EDIT (string1,string2)

Arguments

string1
The first argument, of type string, specifies the string to be edited.

string2
The second argument, of type string, specifies a list of edits to perform, which are separated by commas.

Description

%EDIT is modeled after the OpenVMS DCL lexical function F$EDIT. It is used to perform one or more edits on a specified string. %EDIT processes the string of arguments from left to right. %EDIT gives precedence to the last argument. %EDIT gives precedence to uppercase over lowercase.

The list of edits may contain any combination of the following elements:

Element	Function
COLLAPSE	Removes all tabs and spaces.
COMPRESS	Replaces multiple, consecutive tabs or spaces with a single space.
LOWERCASE	Changes uppercase characters to lowercase.
TRIM	Removes leading and trailing spaces and tabs.
UPCASE	Changes lowercase characters to uppercase.

Examples

Example 1

```
.PRINT "%EDIT(< Fred Smith >, <TRIM,COLLAPSE,UPCASE>)"
```

After lexical processing, the statement apears as the following:

```
.PRINT "FREDSMITH"
```

Example 2

```
.PRINT "%EDIT(<AbCdEfG>,<upcase,lowercase>)
.PRINT "%EDIT(<AbCdEfG>,<lowercase,upcase>)
```

The first source statement produces the string "abcdefg" and the second source statement produces the string "ABCDEFG". Each of the edits in the edit list is performed in sequence, from left to right.

%ELEMENT

Lexical operator for extracting elements from a list of elements.

Format

%ELEMENT (integer,string1,string2)

Arguments

integer

The first argument, of type integer, is the element number to extract. The first element is number 0.

string1

The second argument, of type string, is the delimiter or delimiters that separate elements.

string2

The third argument, of type string, is the list of elements.

Description

%ELEMENT is modeled after the OpenVMS DCL lexical function F$ELEMENT. It is used to extract one element from a string of elements. Note that unlike F$ELEMENT, you may specify multiple delimiters. The result is the specified string element. If the specified element number is greater than the number of elements in the list, the delimiter argument is returned.

Example

```
.PRINT "%ELEMENT (2, <+-*/>, JOE+FRED-TOM*BILL/ERIC)"
```

After lexical processing, the statement appears as:

```
.PRINT "TOM"
```

%EXTRACT

Lexical operator for extracting a range of characters from a string of characters.

Format

%EXTRACT (integer1,integer2,string)

Arguments

integer1
The first argument, of type integer, is the offset at which to begin the extraction. The first character is at offset 0.

integer2
The second argument, of type integer, is the number of characters to extract.

string
The third argument, of type string, is the string from which to extract the characters.

Description

%EXTRACT is modeled after VAX MACRO's %EXTRACT macro string operator and the OpenVMS DCL lexical function F$EXTRACT. %EXTRACT is used to extract a specified range of characters from a string.

Example

```
.PRINT "%EXTRACT(3,4,ABCDEFGHIJKLMNOP)"
```

After lexical processing, the statement appears as:

```
.PRINT "DEFG"
```

%FREG

Lexical operator for obtaining the floating-point register number associated with a symbol.

Format

%FREG (symbol)

Argument

symbol
The single argument, of type string, specifies a symbol that may or may not be currently defined as a floating-point register symbol.

Description

%FREG returns the decimal number of the floating-point register when the specified symbol is defined as a floating-point register symbol. Otherwise, %FREG returns 32.

Example

```
; Is TARG_REG the same as F31?
.IF EQ, <%FREG(TARG_REG)>, <%FREG(31)>
```

If TARG_REG has been defined as floating-point register F5, the statements appear as follows after lexical processing:

```
; Is TARG_REG the same as F31?
.IF EQ, <5>, <31>
```

%INTEGER

Lexical operator for converting the value of an expression to a decimal value.

Format

%INTEGER (integer)

Argument

integer
The single argument, of type integer, is the expression to be converted.

Description

%INTEGER is modeled after the OpenVMS DCL lexical function F$INTEGER. It is used to convert the value of an expression to a decimal value. The result is its decimal value. You can also use %INTEGER to convert a relocatable expression to an absolute value.

Example

```
.PRINT "%INTEGER (<<X+7>*17>)"
```

After lexical processing, if X has the value 3, the statement will appear as:

```
.PRINT "170"
```

%IREG

Lexical operator for obtaining the integer register number associated with a symbol.

Format

%IREG (symbol)

Argument

symbol
The single argument, of type string, specifies a symbol that may or may not be currently defined as an integer register symbol.

Description

%IREG returns the decimal number of the integer register when the specified symbol is defined as an integer register symbol. Otherwise, %IREG returns 32.

Example

```
; Is SRC_REG the same as SP?
.IF EQ, <%IREG(SRC_REG)>, <%IREG(SP)>
```

If SRC_REG has been defined as integer register R16, the statements appear as follows after lexical processing:

```
; Is SRC_REG the same as SP?
.IF EQ, <16>, <30>
```

%LENGTH

Lexical operator for determining the length of a string.

Format

%LENGTH (string)

Argument

string
The single argument, of type string, is the string from which the length is to be computed.

Description

%LENGTH is modeled after VAX MACRO's %LENGTH macro string operator and the OpenVMS DCL lexical function F$LENGTH. %LENGTH is used to determine the length of a string. The result is the length of the string expressed as a decimal number.

Example

```
.PRINT "%LENGTH(<The quick brown fox>)"
```

After lexical processing, the statement appears as:

```
.PRINT "19"
```

%LOCATE

Lexical operator for locating a string of text within another string of text.

Format

%LOCATE (string1,string2)

Arguments

string1
The first argument, of type string, is the string for which %LOCATE searches.

string2
The second argument, of type string, is the string in which the search is performed.

Description

%LOCATE is modeled after VAX MACRO's %LOCATE macro string operator and the OpenVMS DCL lexical function F$LOCATE. %LOCATE is used to locate one string within another. The value returned is the decimal offset to the first occurrence of the first string within the second string. The offset to the first character is 0. If the first string cannot be found within the second string, the length of the second string is returned.

Example

```
.PRINT "%LOCATE (DEF,ABCDEFGHIJKLMNOP)"
```

After lexical processing, the statement appears as:

```
.PRINT "3"
```

%REPEAT

Lexical operator for repeating a specified string a specified number of times.

Format

%REPEAT (integer,string)

Arguments

integer
The first argument, of type integer, is the number of times to repeat the string. If you specify a negative value, the string is repeated 0 times.

string
The second argument, of type string, is the string to be repeated.

Description

%REPEAT is used to repeat the specified string a specified number of times.

Example

```
.PRINT "Never, %REPEAT (3, <ever, >)touch that button!"
```

After lexical processing, the statement appears as:

```
.PRINT "Never, ever, ever, ever, touch that button!"
```

%STRING

Lexical operator for obtaining the value of a lexical string symbol.

Format

%STRING (string)

Argument

string
The single argument is of type string. If the argument is the name of the lexical string symbol, the value of the lexical string symbol is returned. Otherwise, the argument is returned unchanged.

Description

%STRING is modeled after the OpenVMS DCL lexical function F$STRING. %STRING is generally used to obtain the value of a lexical string symbol, but you can use it with any string argument. The lexical substitution operator described in Section 3.4 performs a similar function.

Example

```
FOO = "All the king's horses"
.PRINT "%STRING(FOO)"
```

After lexical processing, the statement appears as:

```
.PRINT "All the king's horses"
```

%TIME

Lexical operator for obtaining the date and time of the assembly unit.

Format

%TIME ()

Description

%TIME is modeled after the OpenVMS DCL lexical function F$TIME. %TIME is used to obtain the time and date of the assembly unit. There are no arguments. The result is a string specifying the date and time of the assembly unit.

Example

```
.PRINT "%TIME()"
```

After lexical processing, the statement appears as:

```
.PRINT " 8-OCT-1991 13:17:57"
```

%TYPE

Lexical operator for obtaining information about a name.

Format

%TYPE (name)

Argument

name
The single argument is of type name. Information is returned about the name specified in the argument.

Description

%TYPE is modeled after the OpenVMS DCL lexical function F$TYPE. %TYPE is used to obtain information about a name. The value returned is a numeric value with certain bit positions, either 0 or 1, depending on whether the specified name has the corresponding attribute. As described elsewhere, the decimal digits of the numeric value are substituted for the %TYPE lexical operator. Table 3–2 shows the symbolic names that are predefined for each attribute.

Table 3–2 %TYPE Attributes

Symbolic Name	Attribute
MACRO64$TYPE_SYMBOL	Name is a numeric symbol name.
MACRO64$TYPE_PROC_DESC	Name is a procedure descriptor name.
MACRO64$TYPE_LABEL	Name is a label.
MACRO64$TYPE_EXTERN	Name is an external name.
MACRO64$TYPE_WEAK	Name is a weak name.
MACRO64$TYPE_PSECT	Name is a psect.
MACRO64$TYPE_MACRO	Name is a macro name.
MACRO64$TYPE_STRING	Name is a lexical string symbol name.
MACRO64$TYPE_OPCODE	Name is an opcode.
MACRO64$TYPE_DIR	Name is a directive.
MACRO64$TYPE_GENREG	Name is a general register.
MACRO64$TYPE_FLTREG	Name is a floating register.

A given name may have zero, one, or several attributes.

Example

```
.macro IS_GR ARG
  .IF equal, %TYPE(ARG) & <MACRO64$TYPE_GENREG>
   .PRINT "ARG is not a general register"
  .ENDC
.endm IS_GR
IS_GR F11
```

Initially, the first line of the IS_GR macro expands as the following:

```
.IF equal, <%TYPE(F11) & MACRO64$TYPE_GENREG>
```

After lexical processing, the statement appears as:

```
.IF equal, <8192 & MACRO64$TYPE_GENREG>
```

In this example, 8192 is the attribute value for a floating-point register. This value could change in subsequent releases. Use only the predefined attribute masks described in Table 3–2. Since the attribute for a general register MACRO64$TYPE_GENREG is 4096, the expression evaluates as 0.

```
<8192 & MACRO64$TYPE_GENREG>
```

4
Macro Arguments

By using macros, you can use a single source statement to insert a sequence of source statements into a program.

A macro definition contains the source statements of the macro. The macro definition may have formal arguments. You can use these formal arguments throughout the sequence of source statements within the definition. When the macro is called, the formal arguments are replaced by the actual arguments within the macro call.

The macro call is a single source statement consisting of the macro name, optionally followed by arguments. When the macro is called, the assembler replaces the line containing the macro call with the source statements in the macro definition. The assembler replaces any occurrences of formal arguments in the macro definition with the actual arguments specified in the macro call. This process is called the macro expansion.

By default, macro expansions are not printed in the assembly listing. To print the macro expansions, you must specify the /SHOW=EXPANSIONS qualifier and argument in the command line. Note that the examples of macro expansions used in this chapter are listed as they would appear using the /SHOW=EXPANSIONS qualifier and argument.

Use .SHOW with a symbolic argument of EXPANSIONS in the source text of a program to specify the listing of expansions.

MACRO–64 provides specific directives for controlling macro execution. For information on these directives, see Table 5–2.

This chapter describes macro arguments and created temporary labels.

4.1 Using Macro Arguments

Macros have two types of arguments: actual and formal. Actual arguments are the text given in the macro call after the name of the macro. Formal arguments are specified by name in the macro definition; that is, after the macro name in the .MACRO directive. Actual arguments in macro calls and formal arguments in macro definitions can be separated by commas (,), tabs, or spaces.

The number of actual arguments in the macro call can be less than or equal to the number of formal arguments in the macro definition. If the number of actual arguments is greater than the number of formal arguments, the assembler displays an error message.

Formal and actual arguments normally maintain a strict positional relationship. That is, the first actual argument in a macro call replaces all occurrences of the first formal argument in the macro definition. This strict positional relationship can be overridden by using keyword arguments. For more information on using keyword arguments, see Section 4.3.

An example of a macro definition using formal arguments follows:

```
.MACRO  STORE  ARG1,ARG2,ARG3
.LONG   ARG1                    ; ARG1 is first argument
.WORD   ARG3                    ; ARG3 is third argument
.BYTE   ARG2                    ; ARG2 is second argument
.ENDM   STORE
```

The following two examples show possible calls and expansions of the macro previously defined:

```
STORE   3,2,1                   ; Macro call
.LONG   3                       ; 3 is first argument
.WORD   1                       ; 1 is third argument
.BYTE   2                       ; 2 is second argument

STORE   X,X-Y,Z                 ; Macro call
.LONG   X                       ; X is first argument
.WORD   Z                       ; Z is third argument
.BYTE   X-Y                     ; X-Y is second argument
```

4.2 Using Default Values

Default values are values that are defined in the macro definition. They are used when no value for a formal argument is specified in the macro call.

Default values are specified in the .MACRO directive as follows:

formal-argument-name = default-value

An example of a macro definition specifying default values follows:

```
.MACRO  STORE   ARG1=12,ARG2=0,ARG3=1000
.LONG   ARG1
.WORD   ARG3
.BYTE   ARG2
.ENDM   STORE
```

The following three examples show possible calls and expansions of the macro defined previously:

```
STORE                   ; No arguments supplied
.LONG   12
.WORD   1000
.BYTE   0

STORE   ,5,X            ; Last two arguments supplied
.LONG   12
.WORD   X
.BYTE   5

STORE   1               ; First argument supplied
.LONG   1
.WORD   1000
.BYTE   0
```

4.3 Using Keyword Arguments

Keyword arguments allow a macro call to specify the arguments in any order. In this case, the macro call must specify the same formal argument names that appear in the macro definition. Keyword arguments are useful when a macro definition has more formal arguments than necessary in the call.

In any one macro call, it is good practice to specify the arguments as either all positional arguments or all keyword arguments. For example, the following macro definition specifies three arguments:

```
.MACRO  STORE   ARG1,ARG2,ARG3
.LONG   ARG1
.WORD   ARG3
.BYTE   ARG2
.ENDM   STORE
```

The following macro call specifies keyword arguments:

```
STORE   ARG3=27+5/4,ARG2=5,ARG1=SYMBL
.LONG   SYMBL
.WORD   27+5/4
.BYTE   5
```

Because the keywords are specified in the macro call, the arguments in the macro call need not be given in the order they were listed in the macro definition.

Positional and keyword arguments may be mixed. Usually, positional arguments are placed before keyword arguments. For example:

```
.MACRO  STORE   ARG1,ARG2,ARG3=27+5/4
.LONG   ARG1
.BYTE   ARG2
.WORD   27+5/4
.ENDM   STORE
```

Note

Keyword arguments are not counted when positional arguments are parsed. This means that when positional and keyword arguments are used in the same macro, one argument can be specified twice. The last value specified for the argument is used.

4.4 Using String Arguments

If an actual argument is a string containing characters that the assembler interprets as separators (such as a tab, space, or comma), the string must be enclosed by delimiters. String delimiters for macro arguments are usually paired angle brackets (<>). A quoted literal enclosed in double quotes ("") is also a valid string argument.

The assembler also interprets any character (except A, B, C, D, O, or X) after an initial circumflex (^) as a delimiter. Note that ^B, ^D, ^O, and ^X are used as radix control operators rather than argument delimiters. ^A is used as the ASCII operator and ^C is used as the complement operator. To pass an angle bracket as part of a string, you can use the circumflex form of the delimiter.

The following are examples of delimited macro arguments:

```
<HAVE THE SUPPLIES RUN OUT?>
<LAB:    CLR     R4>

"A quoted literal is taken as a single parameter value."

^%ARGUMENT IS <LAST,FIRST> FOR CALL%
^?EXPRESSION IS <5+3>*<4+2>?
```

In the last two examples, the initial circumflex indicates that the percent sign (%) and question mark (?) are the delimiters. Note that only the left-hand delimiter is preceded by a circumflex.

The assembler interprets a string argument enclosed by delimiters as one actual argument and associates it with one formal argument. If a string argument that contains separator characters is not enclosed by delimiters, the assembler interprets it as successive actual arguments and associates it with successive formal arguments.

For example, the following macro definition has one formal argument:

```
.MACRO DOUBLE_ASCII STRNG
.ASCII  "STRNG"
.ASCII  "STRNG"
.ENDM   DOUBLE_ASCII
```

The following two macro calls demonstrate actual arguments with and without delimiters:

```
DOUBLE_ASCII <A B C D E>
.ASCII  "A B C D E"
.ASCII  "A B C D E"

DOUBLE_ASCII  A B C D E
%MACRO64-E-TOOMNYARGS, Too many arguments in macro call
```

Note that the assembler interprets the second macro call as having five actual arguments instead of one actual argument with spaces.

When a macro is called, the assembler removes normal delimiters around a string before associating it with the formal arguments. However, a quoted literal within double quotes is treated as a single token and retains its double quote delimiters.

If a string contains a semicolon (;), the string must be enclosed by delimiters; otherwise, the semicolon will mark the start of the comment field. Further, if the string contains a semicolon, you cannot continue the line unless the string is a quoted literal.

You can nest macro invocations, that is, a macro definition can contain a call to another macro. If, within a macro definition, another macro is called and is passed a string argument, you must delimit the argument so that the entire string is passed to the second macro as one argument.

The following macro definition contains a call to the DOUBLE_ASCII macro defined earlier:

```
        .MACRO     CNTDA LAB1,LAB2,STR_ARG
LAB1:   .BYTE      LAB2-LAB1-1            ; Length of 2*string
          DOUBLE_ASCII   <STR_ARG>
   ; Call DOUBLE_ASCII macro
LAB2:
        .ENDM    CNTDA
```

Note that the argument in the call to DOUBLE_ASCII is enclosed in angle brackets even though it does not contain any separator characters. The argument is thus delimited because it is a formal argument in the definition of the macro CNTDA and will be replaced with an actual argument that may contain separator characters.

The following example calls the macro CNTDA, which in turn calls the macro DOUBLE_ASCII:

```
CNTDA  ST,FIN,<LEARN YOUR ABC'S>
ST:     .BYTE   FIN-ST-1
       DOUBLE_ASCII <LEARN YOUR ABC'S>
        .ASCII  "LEARN YOUR ABC'S"
        .ASCII  "LEARN YOUR ABC'S"
FIN:
```

In addition to nested macro invocations, you can also nest macro definitions. That is, you can define one macro within another. In this example, the INNER_MACRO_DEF macro is not defined until the OUTER_MACRO_DEF macro is invoked and expanded:

```
        .macro OUTER_MACRO_DEF
            .macro INNER_MACRO_DEF
                ...
            .endm INNER_MACRO_DEF
        .endm OUTER_MACRO_DEF
```

You can use this capability to define a macro that redefines itself:

```
        .macro SETUP
            A = 75
            B = 92
            C = 87
            D = 0
            E = -12
            F = 42
            .macro SETUP
                ; Setup is done - do nothing
            .endm SETUP
        .endm SETUP
```

In this example, the SETUP macro defines a number of assembly constants. After the SETUP macro has been expanded once, its work is done. Subsequent expansions of the setup macro need not take any action. Therefore, the SETUP macro redefines itself to a macro whose expansion includes only a comment statement. As described elsewhere, when you redefine a macro, the original version of the macro is automatically deleted. If that macro is currently expanding (as would be the case with the previous SETUP macro), the new definition is immediately associated with the macro name. However, the old definition is retained until all pending expansions complete normally. When all pending expansions complete, the old version of the macro is deleted. Thus, the SETUP macro may be invoked any number of times in the assembly unit. Since the first expansion redefines itself, the expansion of the SETUP macro has no effect other than the first time it is invoked.

Another way to pass string arguments in nested macros is to enclose the macro argument in nested delimiters.

NOTE

Each time you use the delimited argument in a macro call, the assembler removes the outermost pair of delimiters before associating it with the formal argument. This method is not recommended because it requires that you know how deeply a macro is nested.

The following macro definition also contains a call to the DOUBLE_ASCII macro:

```
       .MACRO  CNTDA2 LAB1,LAB2,STR_ARG
LAB1:  .BYTE   LAB2-LAB1-1             ; Length of 2*string
       DOUBLE_ASCII  STR_ARG           ; Call DOUBLE_ASCII macro
LAB2:
       .ENDM   CNTDA2
```

Note that the argument in the call to DOUBLE_ASCII is not enclosed in angle brackets.

The following example calls the macro CNTDA2:

```
       CNTDA2 BEG,TERM,<<MIND YOUR P'S AND Q'S>>
BEG:   .BYTE   TERM-BEG-1              ; Length of 2*string
       DOUBLE_ASCII  <MIND YOUR P'S AND Q'S>
       ; Call DOUBLE_ASCII macro
       .ASCII  "MIND YOUR P'S AND Q'S"
       .ASCII  "MIND YOUR P'S AND Q'S"
TERM:
```

Note that even though the call to DOUBLE_ASCII in the macro definition is not enclosed in delimiters, the call in the expansion is enclosed because the call to CNTDA2 contains nested delimiters around the string argument.

4.5 Argument Concatenation

The argument concatenation operator, the apostrophe ('), concatenates a macro argument with constant text or another argument. Apostrophes can either precede or follow a formal argument name in the macro source.

If an apostrophe precedes the argument name, the text before the apostrophe is concatenated with the actual argument when the macro is expanded. For example, if ARG1 is a formal argument associated with the actual argument TEST, then ABCDE'ARG1 is expanded to ABCDETEST.

If an apostrophe follows the formal argument name, the actual argument is concatenated with the text that follows the apostrophe when the macro is expanded. The apostrophe itself does not appear in the macro expansion.

To concatenate two arguments, separate the two formal arguments with two successive apostrophes. Two apostrophes are needed because each concatenation operation discards an apostrophe from the expansion.

An example of a macro definition that uses concatenation follows:

```
       .MACRO CONCAT   A,B
A''B:  .WORD 0
       .ENDM CONCAT
```

Note that two successive apostrophes are used when concatenating the two formal arguments A and B.

An example of a macro call and expansion follows:

```
       CONCAT  X,Y
XY:    .WORD 0
```

4.6 Passing Numeric Values of Symbols

When a symbol is specified as an actual argument, the name of the symbol, not the numeric value of the symbol, is passed to the macro. You can pass the value of the symbol by inserting a backslash (\) before the symbol in the macro call. The assembler passes the characters representing the decimal value of the symbol to the macro. For example, if the symbol COUNT has a value of 2 and the actual argument specified is \COUNT, the assembler passes the string 2 to the macro; it does not pass the name of the symbol, COUNT.

Passing numeric values of symbols is especially useful with the apostrophe (') concatenation operator for creating new symbols.

An example of a macro definition for passing numeric values of symbols follows:

```
        .MACRO WORD n
WORD'n: .WORD n
        .ENDM WORD
```

The following example shows a possible call and expansion of the macro previously defined:

```
       X = 1       ; Start counting at 1
       WORD \X
WORD1: .WORD 1
```

4.7 Using Created Temporary Labels

Temporary labels are often very useful in macros. You can create a macro definition that specifies temporary labels within it, but these temporary labels might be duplicated elsewhere in the temporary label block, possibly causing errors. However, the assembler can create temporary labels in the macro expansion that will not conflict with other temporary labels. These labels are called created temporary labels.

Created temporary labels range from 30000$ to 65535$. Each time the assembler creates a new temporary label, it increments the numeric part of the label name by 1. Consequently, no user-defined temporary labels should be in the range of 30000$ to 65535$.

A created temporary label is specified by a question mark (?) in front of the formal argument name. When the macro is expanded, the assembler creates a new temporary label if the corresponding actual argument is blank. If the corresponding actual argument is specified, the assembler substitutes the actual argument for the formal argument.

The following example is a macro definition specifying a created temporary label:

```
        .MACRO  POSITIVE        ARG1,?L1
        BGE     ARG1,L1
        NEGQ    ARG1,ARG1
L1:    .ENDM    POSITIVE
```

The following three calls and expansions of the macro defined previously show both created temporary labels and a user-defined temporary label:

```
        POSITIVE  R0
        BGE     R0,30000$
        NEGQ    R0,R0
30000$:

        POSITIVE  R5
        BGE     R5,30001$
        NEGQ    R5,R5
30001$:

        POSITIVE  R7,10$
        BGE     R7,10$
        NEGQ    R7,R7
10$:
```

For more information on temporary labels, see Section 2.6.

5
MACRO–64 Assembler Directives

MACRO–64 directives enable you to control certain aspects of the assembly process. This chapter describes the MACRO–64 assembler directives. It describes the various functional categories of directives, as well as a detailed description of each directive. It also discusses the effects of automatic data alignment on assembler directives.

5.1 Program Sections

MACRO–64 allows you to divide your program into sections called psects using the .PSECT directive. Psects are useful for organizing your program, and for low-level control over the linking process. More importantly, each psect falls into one of the following three categories:

- CODE psects can contain only instructions. They contain no data. Psects in this category have the EXE and NOMIX psect attributes.
- DATA psects can contain only data. They contain no instructions. Psects in this category have the NOEXE and NOMIX attributes.
- MIXED psects can contain instructions, data, or both. Psects in this category have the MIX attribute. In addition, they may have either the EXE or NOEXE attribute.

MACRO–64 categorizes psects because:

- There is a significant performance compromise associated with mixing instructions and data in the same program section within the Alpha AXP architecture. This is because the Alpha AXP architecture typically maintains separate memory caches for instructions and data.
- If you mix instructions and data, it is likely that instructions will migrate into the data cache and that data will migrate into the instruction cache. While this situation still yields correct results, the benefits of the instruction and data caches are diminished. Placing data in the instruction stream can also have detrimental effects on the instruction pipeline and the multiple instruction-issue capabilities that most Alpha AXP systems employ.

Since a code psect can contain only instructions and cannot contain arbitrary data, instructions you place in a CODE psect can be analyzed by the assembler optimizer and by the symbolic debugger. Since a mixed psect can contain arbitrary data as well as instructions, instructions you place in a mixed psect are not analyzed by the assembler optimizer or by the symbolic debugger. Instead, the assembler internally converts instructions in a mixed psect to an equivalent data representation.

Because of the compromises associated with mixed psects, by default the assembler creates psects with the NOMIX psect attribute. If you need to place data in a psect that has the EXE attribute, or if you need to place instructions in a psect that has the NOEXE attribute, you must also specify the MIX attribute in the psect's definition. Note that unlike the other psect attributes, the MIX psect attribute is an assembly-time attribute. The MIX psect attribute does not appear in the psect definitions in your object module and it does not affect the linking process.

You can use all assembler directives and instructions within mixed psects. While many assembler directives can be used within both code and data psects, data-storage directives cannot be used in code psects and instructions and instruction-storage directives cannot be used in data psects. If you place instructions or instruction-storage directives in a data psect, or if you place data-storage directives in a code psect, the assembler issues a diagnostic message. Section 5.3 indicates the applicability of each directive within code and data psects.

In summary, code psects may contain only storage for instructions and storage created by instruction directives. Data psects may contain only storage created by data directives. Mixed psects may contain either storage for instructions or data or both. There are no restrictions on the use of data directives in a mixed psect. However, the assembler converts instructions you place in a mixed psect to a data representation. Therefore, instructions in a mixed psect are not analyzed as instructions by either the assembler optimizer or the symbolic debugger. For more information on the .PSECT directive, see Section 5.3.

5.2 Automatic Data Alignment

The assembler can optionally align your data on natural boundaries. While disabled by default, this feature is enabled with the /ALIGNMENT=DATA command-line qualifier or the .ENABLE ALIGN_DATA directive. (See Section 1.2 and Section 5.3.) A natural boundary is an address that is evenly divisibly by the size of the scalar data type being accessed. The MACRO–64 automatic data alignment feature can automatically align word, longword, quadword, and octaword data to a natural boundary. Accessing data on a natural boundary can be significantly faster than an unaligned access. For more information on accessing data on aligned and unaligned boundaries, see the *Alpha Architecture Reference Manual*.

When the MACRO–64 automatic data alignment feature is enabled, the assembler automatically aligns all data-storage directives to a natural boundary. To achieve alignment, the assembler pads with 0 bytes as necessary before allocating the storage for the data directive. Labels that occur before the data-storage directive are defined to be the address of the data storage after alignment. For more information, see Section 2.7.1. The directive descriptions in Section 5.3 indicate those data-storage directives that are affected by automatic data alignment.

5.3 Directives

The general assembler directives provide facilities for performing eleven types of functions. Table 5–1 lists these types of functions and their directives. Some directives are only applicable within data psects (psects with the NOEXE and NOMIX attributes). Other directives are only applicable within code psects (psects with the EXE and NOMIX attributes). All directives are applicable within psects that contain either data and code, or both (psects with the MIX attribute). For information on the MIX assembly-time psect and any associated restrictions, see the description of the .PSECT directive in this chapter.

Table 5–1 shows how each directive can be applied within data or code psects.

Table 5–1 Summary of General Assembler Directives

Category	Directives[1]	Psect Application
Listing control directives	.TITLE .SUBTITLE (.SBTTL) .IDENT .LIST .NLIST .SHOW .NOSHOW .PAGE	All
Message display directives	.PRINT .WARN .ERROR	All
Assembler option directives	.ENABLE (.ENABL) .DISABLE (.DSABL)	All
Data-storage directives	.BYTE .WORD .SIGNED_BYTE .SIGNED_WORD .LONG .ADDRESS .CODE_ADDRESS .LOCAL_CODE_ADDRESS .OCTA .QUAD .ASCII .ASCIC .ASCID .ASCIZ .F_FLOATING (.FLOAT) .D_FLOATING (.DOUBLE) .G_FLOATING .S_FLOATING .T_FLOATING	NOEXE (or MIX)
	.INSTRUCTION	EXE (or MIX)
Location control directives	.ALIGN .BEGIN_EXACT .END_EXACT	All

[1]The alternate form, if any, is given in parentheses.

(continued on next page)

Table 5–1 (Cont.) Summary of General Assembler Directives

Category	Directives[1]	Psect Application
		NOEXE (or MIX)
	.EVEN	
	.ODD	
	.BLKA	
	.BLKB	
	.BLKD	
	.BLKF	
	.BLKG	
	.BLKL	
	.BLKO	
	.BLKQ	
	.BLKS	
	.BLKT	
	.BLKW	
Program sectioning directives	.PSECT	All
	.SAVE_PSECT (.SAVE)	
	.RESTORE_PSECT (.RESTORE)	
Symbol control directives	.DEFINE_FREG	All
	.DEFINE_IREG	
	.EXTERNAL (.EXTRN)	
	.UNDEFINE_REG	
	.WEAK	
Conditional assembly block directives	.IF	All
	.IF_FALSE (.IFF)	
	.ENDC	
	.ELSE	
	.IF_TRUE (.IFT)	
	.IF_TRUE_FALSE (.IFTF)	
	.IIF	
Source inclusion	.INCLUDE	All
Linkage control	.BASE	All
	.END	All
	.LINKAGE_PAIR	NOEXE (or MIX)
	.LOCAL_LINKAGE_PAIR	NOEXE (or MIX)
	.PROCEDURE_DESCRIPTOR	
	.LOCAL_PROCEDURE_ DESCRIPTOR	

[1]The alternate form, if any, is given in parentheses.

The macro directives provide facilities for performing five categories of functions. Table 5–2 lists these categories and their associated directives. See Chapter 4 for detailed information on macro arguments and string operators.

Table 5–2 Summary of Macro Directives

Category	Directives[1]	Application in EXE or NOEXE Psects
Macro definition directives	.MACRO .ENDM .LIBRARY .MCALL	All
Macro deletion directive	.MDELETE	All
Macro exit directive	.MEXIT	All
Repeat block directives	.REPEAT (.REPT) .IRP .IRPC .ENDR	All
Counting directive	.NARG .NCHR	All

[1]The alternate form, if any, is given in parentheses.

The remainder of this chapter describes both the general assembler directives and the macro directives, showing their formats and giving examples of their use. For ease of reference, the directives are presented in alphabetical order.

.ADDRESS

Address storage directive

Format

.ADDRESS address-list

Parameter

address-list
A list of symbols or expressions, separated by commas (,), which MACRO–64 interprets as addresses.

Description

.ADDRESS stores successive quadwords (8 bytes) containing addresses in the object module. Digital recommends that you use .ADDRESS rather than .QUAD for storing address data to provide additional information to the linker.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current (64-bit) boundary before allocating storage.
- You can define a 32-bit address item using macros and the .LONG directive. For example:

```
.macro address_32 item
    .long item
.endm address_32
```

Example

```
.ADDRESS A,B,C
```

.ALIGN

Location counter alignment directive

Format

.ALIGN integer [,fill-specifier]

.ALIGN keyword [,fill-specifier]

Parameters

integer

An integer in the range 0 to 9. The location counter is aligned at an address that is the value of 2 raised to the power of the integer.

keyword

One of five keywords that specify the alignment boundary. The location counter is aligned to an address that is the next multiple of the following values:

Keyword	Size (in Bytes)
BYTE	$2^0 = 1$
WORD	$2^1 = 2$
LONG	$2^2 = 4$
QUAD	$2^3 = 8$
OCTA	$2^4 = 16$

[,fill-specifier]

Any expression that resolves to an assembly-time integer value containing no forward references. The filling is done per byte, regardless of the alignment. If the value you specify is not in the range of 0 to 255, the assembler issues a diagnostic message and truncates the value.

Description

.ALIGN aligns the location counter to the boundary specified by either an integer or a keyword.

Notes

- If .ALIGN is specified in a psect with the EXE and NOMIX attributes, the fill-specifier is ignored. The assembler aligns the psect to the requested boundary padding with NOP or FNOP instructions.
- If .ALIGN is specified in a psect that does not have the EXE attribute and a fill-specifier is specified, the assembler aligns the psect to the requested boundary padding, with byte locations using the fill-specifier as the initial value for the generated byte padding.
- If the fill-specifier expression encounters a value that is too large to fit in a boundary specified by the keyword, the data is truncated and an informational message is displayed.

- The alignment that you specify in .ALIGN cannot exceed the alignment of the psect in which the alignment is attempted (see the description of .PSECT). For example, if you are using the BYTE psect alignment and you specify .ALIGN with a word or larger alignment, the assembler displays an error message.

Examples

Example 1

```
   .PSECT A,QUAD  ; Begin at quadword
B::.BYTE 4        ; Data is byte
   .ALIGN QUAD    ; Next data is
C::.WORD 6        ; also quadword aligned
```

Example 2

```
   .PSECT A,EXE,NOMIX,OCTA
L1::TRAPB:        ; offset 0
   .ALIGN OCTA    ; NOP padding bytes 4..15
    TRAPB:         ; offset 16
```

Example 3

```
   .PSECT A,NOEXE,NOMIX,OCTA
L1:.WORD 5        ; byte offset 0..1
   .ALIGN QUAD,2  ; fill specifier initial value
                  ; of 2 for bytes 2..7
   .WORD 6        ; byte offsets 8..9
```

.ASCIC

Counted ASCII string storage directive

Format

.ASCIC quoted-literal

Parameter

quoted-literal
An ASCII string delimited with double quotes.

Description

.ASCIC performs the same function as .ASCII, except that .ASCIC inserts a count byte before the string data. The count byte contains the length of the string in bytes. The length given includes any bytes of nonprintable characters specified using the backslash (\) operator, but excludes the count byte. See Section 2.4 for a description of how to specify quoted literals.

.ASCIC is useful in copying text because the count indicates the length of the text to be copied.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.ASCIC "MY STRING"  ; In the listing, this becomes:
                    ; .BYTE 9
                    ; .ASCII \MY STRING\
```

.ASCID

String-descriptor ASCII string storage directive

Format

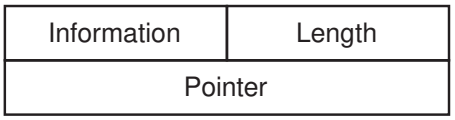
.ASCID quoted-literal

Parameter

quoted-literal
An ASCII string delimited with double quotes. For more information on how to specify quoted literals, see Section 2.4.

Description

.ASCID performs the same function as the .ASCII directive, except that .ASCID inserts a string descriptor before the string data. The descriptor format is identical to that defined for OpenVMS AXP and OpenVMS VAX. The string descriptor has the following format:



ZK–6462A–GE

Notes

- String descriptors are used in calling certain system routines.
- The string-length field is two bytes in size.
- Descriptor information (2 bytes) is always set to ^X010E.
- The pointer field is a 32-bit pointer to the string (4 bytes).
- If natural alignment is enabled (using .ENABLE ALIGN_DATA), the descriptor is quadword aligned. This allows you to access the entire descriptor (2 data words and a longword address) on a quadword boundary.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Examples

Example 1

```
.DESCR1:   .ASCID "ARGUMENT FOR CALL"     ; String descriptor
```

Example 2

```
.DESCR2:   .ASCID "SECOND ARGUMENT"  ; Another string descriptor
```

.ASCII

ASCII string storage directive

Format

.ASCII quoted-literal

Parameters

quoted-literal
An ASCII string delimited with double quotes.

Description

.ASCII stores the ASCII value of each character in the ASCII string or the value of each byte expression in the next available byte. See Section 2.4 for a description of how to specify quoted literals.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Examples

```
.ASCII "MY STRING"
```

.ASCIZ

Zero-terminated ASCII string storage directive

Format

.ASCIZ quoted-literal

Parameter

quoted-literal
An ASCII string delimited with double quotes.

Description

.ASCIZ performs the same function as .ASCII, except that .ASCIZ appends a null byte as the final character of the string. When a list or text string is created with an .ASCIZ directive, you need only perform a search for the null character in the last byte to determine the end of the string. See Section 2.4 for a description of how to specify quoted literals.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.ASCIZ "MY STRING"    ; Equivalent to
                      ; .ASCII "MY STRING \x00"
```

.BASE

Base register directive

Format

.BASE Rn [,base_expression]

Parameters

Rn

One of the base registers, R0 through R30, FP, and SP.

base_expression

The base address, which is optional, and can be one of the following:

- An absolute expression
- A relocatable expression
- An external expression

An expression must not contain forward references or implicit external symbols. An implicitly defined external symbol is a symbol that the assembler defaults to an external symbol. This occurs when the assembler encounters references to a symbol, but does not encounter a definition for the symbol or an .EXTERNAL directive that declares the symbol.

Description

The .BASE directive is used to inform the assembler that a specified base register contains a specified base address. Later in your program, the assembler allows you to implicitly reference the specified base register. When the assembler knows which base addresses are stored in one or more base registers, it can convert an expression to an offset from one of the base registers previously specified in a .BASE directive. .BASE provides a convenient and more readable shorthand for accessing memory and constant values using base registers. .BASE also makes it easier for you to change your register assignments if you later modify your code.

The base expression is optional. If the base expression is specified, this base address value is assumed by the assembler to be in the specified register, Rn. If the base expression is omitted, the contents of the specified base register, Rn, is considered undefined until a new base expression is associated with the base register.

R31 is defined to always contain 0, according to the architecture definition. Therefore, R31 is known to be a predefined base register containing 0. For every assembly, the assembler assumes the following statement:

```
.BASE R31, 0
```

Because the contents of R31 cannot change, you cannot specify a base address for R31.

You can use the .BASE directive to implicitly reference base registers. You can also automatically compute offsets from a base address known to be in a register to a base address you use in an instruction argument.

Assembler Directives
.BASE

Most of the memory format Alpha AXP instructions are defined such that one of their arguments must have a base register and an offset. If the assembler encounters only an expression with no base register, the assembler attempts to find a base register that contains a base address or constant within a 16-bit signed offset of the value of the expression. If it finds such a base register, the assembler computes an offset that, when added to the value of the base register, results in a value equal to the expression specified in the instruction argument.

Examples

Example 1

```
.EXTERNAL COMM_AREA          ❶
.BASE R1, COMM_AREA          ❷
CURR_LINE       = COMM_AREA + 0
CURR_COLUMN     = COMM_AREA + 4
CURR_MODE       = COMM_AREA + 8
LDA     R4, 17                  ; LDA R4, 17(R31)   ❸
LDL     R2, CURR_LINE           ; LDL R2, 0(R1)     ❹
LDL     R3, CURR_COLUMN         ; LDL R3, 4(R1)
STL     R4, CURR_MODE           ; STL R4, 8(R1)
```

❶ This statement declares an external symbol, COMM_AREA. COMM_AREA is a global symbol that represents the base address of a three-longword communication area that is used by different routines in the program.

❷ This statement informs the assembler that base register R1 contains the base address, COMM_AREA, of this communication area. The next three statements define variables within the communication area.

❸ The first instruction shows how you can load registers with constant values in the range −32,768 to +32,767 by implicitly using R31 as the base register.

❹ The last three statements show how the .BASE directive allows you to implicitly reference base registers and automatically compute offsets. In each of these instructions, the second argument is defined to require an offset and a base register.

Since no base register is specified, the assembler attempts to imply the base register and compute the offset based upon information given in previous .BASE directives.

In the last three instructions, the address argument is within −32,768 to +32,767 of the base address known to be in R1 (that is, COMM_AREA). Therefore, R1 is selected as the base register. The assembler also computes the correct offset from the base address known to be in R1 to the address specified in the instruction argument.

Example 2

The assembler performs a sequential search through the list of possible base registers, R0 through R31. It uses the first definition possible if multiple base registers are valid. For example:

```
.BASE R5, 300
:
LDQ  R10, 100
```

The assembler outputs the LDQ instruction as follows:

```
LDQ  R10, -200(R5)
```

Both R31 and R5 are defined as base registers that can be used in constructing the instruction argument. R31 always contains 0. In this example, R5 is also known to contain the constant 300. The assembler uses the first base register, starting at R0 and progressing to R31, which provides a known value within −32,768 to +32,767 of the specified argument value. Since the assembler considers R5 before it considers R31, R5 is used rather than R31.

.BEGIN_EXACT

Exact instruction block directive

Format

.BEGIN_EXACT

Description

An exact instruction block suppresses code optimizations (SCHEDULE and PEEPHOLE) regardless if these optimizations are enabled for the assembly unit. Unlike .ENABLE and .DISABLE, which can be used to enable or disable specific optimizations for the entire assembly unit, .BEGIN_EXACT and .END_EXACT allow you to suppress optimization for a specified range of instructions. Instructions outside the specified range remain subject to any optimizations you have enabled.

Notes

- This directive cannot appear in a psect with the NOEXE and NOMIX attributes.
- Although this directive is accepted by the assembler in a psect with the MIX attribute, it has no effect in these psects since no code optimizations are in affect for MIX psects.
- .BEGIN_EXACT must be paired with a matching .END_EXACT to close the exact instruction block.
- .BEGIN_EXACT and .END_EXACT instruction blocks can be nested. The outermost level of the .BEGIN_EXACT and matching .END_EXACT directives delimit the actual exact instruction block from which code optimizations are suppressed. Nesting .BEGIN_EXACT and .END_EXACT instruction blocks can be useful in macro definitions where the macro expansion requires an exact instruction sequence. Nested .BEGIN_EXACT and .END_EXACT instruction blocks allow a macro to be invoked both from within and without the exact instruction block.
- .BEGIN_EXACT does not affect automatic alignment. Automatic alignment is enabled with the .ENABLE ALIGN_CODE and .ENABLE ALIGN_DATA directives or with the /ALIGN=(CODE,DATA) command-line qualifier.

Examples

The following example shows an instruction sequence prior to optimization:

```
addf f7, f8, f9     ; 1
addf f2, f3, f4     ; 2
addl r5, r6, r7     ; 3
addl r8, r9, r10    ; 4
```

The assembler optimizes the previous example to a sequence similar to the following instruction sequence:

```
        :
addf f7, f8, f9     ; 1
addl r5, r6, r7     ; 3
addf f2, f3, f4     ; 2
addl r8, r9, r10    ; 4
        :
```

If you choose to suppress optimization in the previous example, enclose the four instructions with the .BEGIN_EXACT and .END_EXACT directives, as shown in the following example:

```
.BEGIN_EXACT
addf f7, f8, f9     ; 1
addf f2, f3, f4     ; 2
addl r5, r6, r7     ; 3
addl r8, r9, r10    ; 4
.END_EXACT
```

.BLK*x*

Block storage allocation directives

Format

.BLKA [expression]

.BLKB [expression]

.BLKD [expression]

.BLKF [expression]

.BLKG [expression]

.BLKL [expression]

.BLKO [expression]

.BLKQ [expression]

.BLKS [expression]

.BLKT [expression]

.BLKW [expression]

Parameter

expression
An integer expression specifying the amount of storage to be allocated. All the symbols in the expression must be defined at the current point in the assembly and the expression must be an absolute expression. If the expression is omitted, a default value of 1 is assumed.

Description

MACRO–64 has the following 11 block storage directives:

Directive	Reserves Storage for:	Bytes Allocated
.BLKA	Addresses (quadwords)	8 * value of expression
.BLKB	Byte data	Value of expression
.BLKD	Double-precision floating-point data (quadwords)	8 * value of expression
.BLKF	Single-precision floating-point data (longwords)	4 * value of expression
.BLKG	G_floating data (quadwords)	8 * value of expression
.BLKL	Longword data	4 * value of expression
.BLKO	Octaword data	16 * value of expression
.BLKQ	Quadword data	8 * value of expression
.BLKS	S_floating data (longwords)	4 * value of expression
.BLKT	T_floating data (quadwords)	8 * value of expression
.BLKW	Word data	2 * value of expression

Each directive reserves storage for a different data type. The value of the expression determines the number of data items for which MACRO–64 reserves storage. For example, .BLKL 4 reserves storage for 4 longwords of data and .BLKB 2 reserves storage for 2 bytes of data. The total number of bytes reserved is equal to the length of the data type times the value of the expression.

Notes

- If automatic data alignment is enabled, the .BLKx directives align the current location counter to one of the alignments listed in the following table:

Directive	Alignment
.BLKA	Quadword (64-bit) boundary
.BLKB	None
.BLKD	Quadword (64-bit) boundary
.BLKF	Longword (32-bit) boundary
.BLKG	Quadword (64-bit) boundary
.BLKL	Longword (32-bit) boundary
.BLKO	Octaword (128-bit) boundary
.BLKQ	Quadword (64-bit) boundary
.BLKS	Longword (32-bit) boundary
.BLKT	Quadword (64-bit) boundary
.BLKW	Word (16-bit) boundary

- You can only use these directives within psects having either the NOEXE or MIX attributes. For more information on psects, see Section 5.1.

Example

```
    .PSECT A,NOEXE
B:: .BLKW  10             ; 10 words (20 bytes) of storage
    .BLKQ   5             ; 5 quadwords (40 bytes) of storage

    .BLKW                 ; 1 word (2 bytes) of storage
```

.BYTE

Byte storage directive

Format

.BYTE expression-list

Parameter

expression-list
One or more expressions separated by commas. Each expression is first evaluated as a quadword expression; then the value of the expression is truncated to fit in a byte. The value of each expression should be in the range 0 to 255 for unsigned data or in the range −128 to +127 for signed data.

Description

.BYTE generates successive bytes of binary data in the object module.

Notes

- The assembler displays a warning message if the expression is outside the range −128 to −255.
- The assembler will truncate the most significant bits of an integer or external value that is too large to store in eight bits.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.

Example

```
A:      .BYTE   5       ; Stores 5 in a byte
```

.CODE_ADDRESS

Code address storage directive

Format

.CODE_ADDRESS name-list

Parameter

name-list
A list of symbols separated by commas. These symbols should reference either a procedure descriptor name, such as a routine name, or an externally defined procedure descriptor.

Description

.CODE_ADDRESS causes the code addresses of the specified identifiers to be placed at the current psect and current location counter. The specified identifier should reference a procedure descriptor defined in the image.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.

Example

```
.CODE_ADDRESS A
```

.D_FLOATING
.DOUBLE

Floating-point storage directive

Format

.D_FLOATING floating-point-number-list

.DOUBLE floating-point-number-list

Parameter

floating-point-number-list
A comma-separated list of floating-point constants. The constants cannot contain any operators except unary plus or unary minus. For more information on floating-point numbers, see Section 2.3.2.

Description

.D_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .D_FLOATING generates 64-bit, double-precision floating-point data (1 bit of sign, 8 bits of exponent, and 55 bits of fraction). See the description of .F_FLOATING for information on storing single-precision floating-point numbers and the descriptions of .G_FLOATING, .S_FLOATING, and .T_FLOATING for descriptions of other floating-point constants.

Notes

- Double-precision floating-point numbers are always rounded.
- The alternate form of .D_FLOATING is .DOUBLE.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.
- The Alpha AXP architecture supports only conversion operations for the D floating-point data type.

Example

```
.D_FLOATING  3.1E+02
```

.DEFINE_FREG

Define floating-point register symbol directive

Format

.DEFINE_FREG regsym regnum

Parameters

regsym
A MACRO–64 identifier.

regnum
An assembly-time expression or a currently defined register symbol.

Description

The identifier you specify as the first argument to .DEFINE_FREG becomes a floating-point register symbol. Thereafter, it cannot be used as a MACRO–64 identifier. Specifically, the user-defined register symbol is only allowed where a register is allowed. In this sense, the user-defined register symbol is reserved until the end of the assembly unit or until you delete its definition (see .UNDEFINE_REG), whichever occurs first.

The second argument to .DEFINE_FREG can be either an integer expression or a currently defined floating-point register symbol. An integer expression indicates the register number to assign to the register symbol. The expression can contain no forward or external references and must be in the range of 0 to 31. Alternatively, you can define a register symbol in terms of another currently defined register symbol. A currently defined register symbol is a predefined register symbol or a register symbol that you have previously defined. In this case, the new register symbol you specify with the first argument receives the current value of the register symbol you specify with the second argument.

Notes

- You cannot define a floating-point register in terms of an integer register and vice versa.
- You cannot redefine a currently defined register symbol with a different register number. To redefine a register symbol, you must first delete the old definition with the .UNDEFINE_REG directive.

Example

```
.DEFINE_IREG A0     16      ; A0 is integer register 16
.DEFINE_IREG A1     R17     ; A1 is integer register 17,
                            ;  defined in terms of the
                            ;  predefined R17 register symbol
.DEFINE_IREG PTR    A0      ; PTR is integer register 16,
                            ;  defined in terms of the
                            ;  previously-defined A0 register
                            ;  symbol
.DEFINE_FREG $F0    0       ; $F0 is floating register 0
.DEFINE_FREG $F1    F1      ; $F1 is floating register 1,
                            ;  defined in terms of the
                            ;  predefined F1 register symbol
.DEFINE_FREG RADIUS $F1     ; RADIUS is floating register 1,
                            ;  defined in terms of the
                            ;  previously defined $F1
                            ;  register symbol
.DEFINE_IREG X1     R5      ; X1 is integer register 5,
                            ;  defined in terms of the
                            ;  predefined R5 register symbol
.DEFINE_IREG X1     5       ; 2nd definition is the same
                            ;  value, so no diagnostic
                            ;  results
.DEFINE_IREG X1     7       ; Warning: redefinition with a
                            ;  different value
.DEFINE_IREG X2     F5      ; Error: cannot define an integer
                            ;  register in terms of a
                            ;  floating register

LDQ         R1, (PTR)       ; LDQ R1, (R16)
LDG         RADIUS, (A1)    ; LDG F1, (R17)
```

.DEFINE_IREG

Define integer register symbol directive

Format

.DEFINE_IREG regsym regnum

Parameters

regsym
A MACRO–64 identifier.

regnum
An assembly-time expression or a currently defined register symbol.

Description

The identifier you specify as the first argument to .DEFINE_IREG becomes an integer register symbol. Thereafter, it cannot be used as a MACRO–64 identifier. Specifically, the user-defined register symbol is only allowed where a register is allowed. In this sense, the user-defined register symbol is reserved until the end of the assembly unit or until you delete its definition (see .UNDEFINE_REG), whichever occurs first.

The second argument to .DEFINE_IREG can be either an integer expression or a currently defined integer register symbol. An integer expression indicates the register number to assign to the register symbol. The expression can contain no forward or external references and must be in the range of 0 to 31. Alternatively, you can define a register symbol in terms of another, currently defined register symbol. A currently defined register symbol is a predefined register symbol or a register symbol that you have previously defined. In this case, the new register symbol you specify with the first argument receives the current value of the register symbol you specify with the second argument.

Notes

- You cannot define an integer register in terms of a floating-point register and vice versa.
- You cannot redefine a currently defined register symbol with a different register number. To redefine a register symbol, you must first delete the old definition with the .UNDEFINE_REG directive.

Example

Refer to the example in .DEFINE_FREG.

.DISABLE
.DSABL

Disable assembler functions directive

Format

.DISABLE argument-list

.DSABL argument-list

Parameter

argument-list
One or more of the symbolic arguments listed in Table 5–3 under the description of .ENABLE. You can use either the long or the short form of the symbolic arguments. If you specify multiple arguments, separate them by commas, spaces, or tabs.

Description

.DISABLE disables the specified assembler function. See the description of .ENABLE for more information.

The alternate form of .DISABLE is .DSABL.

.ELSE

Conditional assembly block directive

Format

.ELSE

Description

A conditional assembly block is a series of source statements that is assembled only if a certain condition is met. .IF starts the conditional block and .ENDC ends the conditional block; each .IF must have a corresponding .ENDC. The .IF directive contains a condition test and one or two arguments. The condition test specified is applied to the arguments. If the test is met, all MACRO–64 statements between .IF and .ELSE are assembled. If the test is not met, the statements between .ELSE and .ENDC are assembled.

Conditional blocks can be nested; that is, a conditional block can be inside another conditional block. In this case, the statements in the inner conditional block are assembled only if the condition is met for both the outer and inner block. For more information, see the description of the .IF directive.

Notes

- You cannot use the .ELSE directive in the same conditional block as the .IF_x directive.
- The .ELSE directive is similar to the .IF_FALSE directive. However, you can only use .ELSE once within a conditional block. .IF_FALSE can be used any number of times in a conditional block.

Example

Here is an example of a conditional assembly directive:

```
.IF EQUAL  ALPHA+1        ; Assemble block if ALPHA+1=0.
  .
  .
.ELSE                     ; Assemble when .IF=false.
  .
  .
.ENDC
```

.ENABLE
.ENABL

Enable assembler functions directive

Format

.ENABLE argument-list

.ENABL argument-list

Parameter

argument-list
One or more of the symbolic arguments listed in Table 5–3. You can use either the long form or the short form of the symbolic arguments.

If you specify multiple arguments, separate them with commas, spaces, or tabs.

Table 5–3 .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
ALIGN_CODE		Disabled	The code alignment option aligns certain branch target labels. The ALIGN_CODE option is disabled for the assembly unit if any one of the following is true: • /NOALIGNMENT=CODE is specified on the command line. • .DISABLE ALIGN_CODE is specified in the source program. • The /ALIGNMENT=CODE option is defaulted. See Section B.7 for information on optimizations and automatic alignment.

(continued on next page)

Table 5–3 (Cont.) .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
ALIGN_DATA		Disabled	When ALIGN_DATA is disabled, the data-storage directives put each succeeding item on the next byte boundary. When ALIGN_DATA is enabled, the data-storage directives put each succeeding item on natural boundaries (such as words on word boundaries, longwords on longword boundaries) and add pad bytes as necessary. Accessing data on anything other than natural boundaries usually incurs a significant performance penalty. You can enable or disable the ALIGN_DATA option for specific ranges within your program. For more information on automatic data alignment, see Section 5.2. Also, see Section B.7 for information on optimizations and automatic alignment.
FLOAT		Enabled	Controls whether the assembler generates floating-point instructions when optimizing code and performing code-label alignment. Currently, the only floating-point instruction generated by the assembler during optimization and alignment processing is FNOP, the floating-point no-operation instruction. If you specify .DISABLE FLOAT, the assembler does not generate any floating-point instructions as part of optimization and alignment processing. The initial value of this option is specified by the /ENVIRONMENT=[NO]FLOAT command-line option. The last value of the FLOAT option at the end of assembly determines whether FLOAT is enabled or DISABLED.

(continued on next page)

Table 5–3 (Cont.) .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
GLOBAL	GBL	Enabled	When GLOBAL is enabled, the assembler implicitly treats any undefined symbol as an external reference defined in another module. If the GLOBAL option is disabled, the assembler issues a warning and implicitly treats the undefined symbol as an external reference assumed to be defined in another module. The last value of the GLOBAL option at the end of assembly determines whether the GLOBAL option is enabled or disabled.
LOCAL_BLOCK	LSB	Disabled	Used to override the default assembler behavior to define a temporary label block. (A temporary label is of the form *n*$ where *n* represents a number.) A temporary label block is usually delimited by two user-defined local or global labels. However, the .ENABLE LOCAL_BLOCK directive defines the start of a block that is terminated by one of the following: • A second .ENABLE LOCAL_ BLOCK directive. • A .DISABLE LOCAL_BLOCK directive followed by a user-defined local or global label, or a .PSECT directive.
PEEPHOLE		Disabled	Peephole optimization reduces the strength of certain instructions and eliminates instructions where possible. The PEEPHOLE option is disabled for the assembly unit if any one of the following is true: • /NOOPTIMIZE=PEEPHOLE is specified on the command line. • .DISABLE PEEPHOLE is specified in the source program. • The PEEPHOLE option is defaulted. See Section B.7 for information on optimizations and automatic alignment.

(continued on next page)

Table 5–3 (Cont.) .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
PREPROCESSOR_ OUTPUT		Enabled	When PREPROCESSOR_OUTPUT is enabled, the MACRO–64 preprocessor processes your source statements and outputs to the preprocessor-output file. If the PREPROCESSOR_ OUTPUT option is disabled, your source statements are processed as before. However, output of these statements to the preprocessor-output file is suppressed. Neither .ENABLE PREPROCESSOR_ OUTPUT nor .DISABLE PREPROCESSOR_OUTPUT is passed through to the preprocessor-output file. These two directives have no effect unless you specify /PREPROCESSOR_ ONLY on the command line. For more information, see the description of the /PREPROCESSOR_ONLY command-line qualifier in Chapter 1.
SCHEDULE		Disabled	Instruction scheduling optimization reorders instructions to more optimally utilize the instruction pipeline. The SCHEDULE option is disabled for the assembly unit if any one of the following is true: • /NOOPTIMIZE=SCHEDULE is specified on the command line. • .DISABLE SCHEDULE is specified in the source program. • The SCHEDULE option is defaulted. See Section B.7 for information on optimizations and automatic alignment.

Description

.ENABLE enables the specified assembly function. .ENABLE and its negative form, .DISABLE, control the following assembler functions:

- Creating local label blocks
- Specifying that undefined symbol references are external references
- Enabling or disabling specific optimizations for the assembly unit

You can enable one or more specific optimization options with either the .ENABLE directive or the /OPTIMIZE command-line qualifier, or both. For more information on command-line qualifiers, see Section 1.2. See Section B.7 for information on optimizations. If you explicitly disable one or more specific optimization options with the .DISABLE directive, those optimization options are disabled regardless of the command-line options you specify.

Notes

- The alternate form of .ENABLE is .ENABL.

Examples

Example 1

The following example shows the ALIGN_DATA option:

```
.PSECT A, NOEXE
.ENABLE ALIGN_DATA  ; Align on natural
                    ; natural boundaries
A: .BYTE 1          ;
B: .QUAD 1000       ; B is allocated at
                    ; a natural boundary -
                    ; specifically at A + 7
.DISABLE ALIGN_DATA ;
C: .BYTE 2          ;
D: .QUAD 1001       ; D is allocated at
                    ; an unaligned boundary -
                    ; specifically C + 1
```

Example 2

The following example shows the GLOBAL option disabled:

```
.DISABLE GLOBAL
.ADDRESS X         ; Assembler issues a warning
.END
```

Example 3

The following example shows the LOCAL_BLOCK option enabled:

```
      .ENABLE LOCAL_BLOCK

      .PSECT A,NOEXE
A1::
5$:   .PROCEDURE_DESCRIPTOR PROC_1   ; Temporary label 5$
      .blkb 32
A2::
      .address 5$         ; By default the declaration
                          ; of A2 would have ended the
                          ; temporary label block and
                          ; made this reference to 5$
                          ; illegal.  However, this default
                          ; behavior has been overridden
                          ; by the use of .ENABLE LOCAL_BLOCK.

      .DISABLE LOCAL_BLOCK
      .END
```

Example 4

The following example shows an unoptimized and optimized list of instructions with the SCHEDULE and PEEPHOLE options enabled:

```
.ENABLE PEEPHOLE,SCHEDULE
.psect A,EXE,QUAD     ; unoptimized
TRAPB
A::ADDF F1,F2,F3
   ADDF F4,F5,F6
   ADDL R1,R2,R3
   ADDL R4,R5,R6
```

This example shows the optimized list of instructions:

```
.ENABLE PEEPHOLE,SCHEDULE
.psect A,EXE,QUAD     ; optimized
A::ADDF F1,F2,F3
   ADDL R1,R2,R3
   ADDF F4,F5,F6
   ADDL R4,R5,R6
```

The following example shows a repeat block that initializes a block of 1000 longwords to the values of 0 through 999. The .DISABLE PREPROCESSOR_OUTPUT directive suppresses from the preprocessor output file those statements that are incompatible with the OSF/1 Assembler.

```
.DISABLE PREPROCESSOR_OUTPUT
 I=0
.REPEAT 1000
   .ENABLE PREPROCESSOR_OUTPUT
   .LONG %INTEGER(I)
   .DISABLE PREPROCESSOR_OUTPUT
   I = I + 1
.ENDR
```

.END

Assembly termination directive

Format

.END [label]

Parameter

label
The procedure descriptor name that specifies the routine (called the transfer address) where program execution begins. This argument is optional.

Description

.END terminates the source program. No additional text should occur beyond this point in the current source file, or in any additional source files specified in the command line for this assembly. If any additional text does occur, the assembler ignores it. The additional text does not appear in the listing file nor does it affect the object file.

Notes

- When an executable image consisting of several object modules is linked, only one object module should be terminated by an .END directive that specifies a transfer address. All other object modules should be terminated by .END directives that do not specify a transfer address. If an executable image contains either no transfer address or more than one transfer address, the linker displays an error message.
- For more information, see the .PROCEDURE_DESCRIPTOR directive.

Example

```
  .
  .
  .
.PROCEDURE_DESCRIPTOR TRANSFER1,code_address_T1
  .
  .
  .
.END TRANSFER1        ; TRANSFER1 is module transfer address
```

.ENDC

End conditional directive

Format

.ENDC

Description

.ENDC terminates the conditional range started by the .IF directive. See the description of .IF for more information and examples.

.ENDM

End macro definition directive

Format

.ENDM [macro-name]

Parameter

macro-name
The name of the macro whose definition is to be terminated. The macro name is optional; if specified, it must match the name defined in the matching .MACRO directive. The macro name should be specified so that the assembler can detect any improperly nested macro definitions.

Description

.ENDM terminates the macro definition. See the description of .MACRO for an example that uses an .ENDM directive.

Notes

- If .ENDM is encountered outside a macro definition, the assembler displays an error message.

.ENDR

End repeat range directive

Format

.ENDR

Description

.ENDR indicates the end of a repeat range. It must be the final statement of every repeat block. A repeat block consists of any range of text beginning with the .IRP, .IRPC, or .REPEAT directive. For more information, see the description for the .IRP, .IRPC, or .REPEAT directives for examples of how to use the .ENDR directive.

Notes

- If .ENDR is encountered outside a repeat block, the assembler displays an error message.

.END_EXACT

End exact instruction block directive

Format

.END_EXACT

Description

.END_EXACT delimits the end of an exact instruction block. An exact instruction block suppresses the SCHEDULE and PEEPHOLE optimizations for the specified range of instructions regardless if code optimizations are enabled for the assembly unit.

For more information on the .END_EXACT directive, see the description of the .BEGIN_EXACT directive in this chapter.

.ERROR

Error directive

Format

.ERROR quoted-literal

Parameter

quoted-literal
A string of characters, between a pair of double quotes, displayed during assembly. For more information on quoted literals, see Section 2.4.

Description

.ERROR causes the assembler to display an error message on the terminal or batch log file and in the listing file (if there is one).

Using .ERROR prevents an output object file from being produced.

Notes

- .PRINT, .WARN, and .ERROR are directives used to display messages. You can use them to display information indicating unexpected or important conditions within the assembly.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.ERROR "Illegal Arguments"
^
%MACRO64-E-GENERROR, Generated ERROR: Illegal Arguments
at line number 3 in file DISK$:[TEST]ERROR.M64;2
```

.EVEN

Even location counter alignment directive

Format

.EVEN

Description

.EVEN ensures that the current value of the location counter is even by adding 1 if the current value is odd. If the current value is already even, no action is taken.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.

.EXTERNAL
.EXTRN

External symbol attribute directive

Format

.EXTERNAL symbol-list

.EXTRN symbol-list

Parameter

symbol-list
A list of symbol names separated by commas.

Description

.EXTERNAL indicates that the specified symbols are external; that is, the symbols are defined in another object module.

Notes

- The alternate form of .EXTERNAL is .EXTRN.

Example

```
        .EXTERNAL B     ; B is defined in another module
           .
           .
           .
A::     .ADDRESS B     ; Its address is stored here
```

.F_FLOATING
.FLOAT

Floating-point storage directive

Format

.F_FLOATING floating-point-number-list

.FLOAT floating-point-number-list

Parameter

floating-point-number-list
A list of one or more floating-point constants separated by commas. For more information on floating-point numbers, see Section 2.3.2. The constants cannot contain any operators except unary plus or unary minus.

Description

.F_FLOATING evaluates the specified floating-point constant(s) and stores the results in the object module. .F_FLOATING generates 32-bit, single-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 23 bits of fractional significance). See the description of .D_FLOATING for information on storing double-precision floating-point constants and the descriptions of .G_FLOATING, S_FLOATING, and T_FLOATING for descriptions of other floating-point constants.

Notes

- The alternate form of .F_FLOATING is .FLOAT.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a longword (32-bit) boundary before allocating storage.

Example

```
.F_FLOATING 1.0,3.0E+2
```

.G_FLOATING

G_floating-point storage directive

Format

.G_FLOATING floating-point-number-list

Parameter

floating-point-number-list
A comma-separated list of one or more floating-point constants. For more information on floating-point numbers, see Section 2.3.2. The constants cannot contain any operators except unary plus or unary minus.

Description

.G_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .G_FLOATING generates 64-bit data (1 bit of sign, 11 bits of exponent, and 52 bits of fraction). See the description of .D_FLOATING for information on storing double-precision floating-point constants and the descriptions of .F_FLOATING, S_FLOATING, and T_FLOATING for descriptions of other floating-point constants.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.

Example

```
.G_FLOATING  2.0E-3
```

.IDENT

Identification directive

Format

.IDENT quoted-literal

Parameter

quoted-literal
A 1- to 31-character string, within double quotes, that identifies the module, such as a string that specifies a version number. For more information on quoted literals, see Section 2.4.

Description

.IDENT provides a means of identifying the object module. This identification is in addition to the name assigned to the object module with .TITLE. You can specify a character string in .IDENT to label the object module. This string is printed in the header of the listing file and also appears in the object module.

Notes

- If a source module contains more than one .IDENT, the last directive given establishes the character string that forms part of the object module identification.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.IDENT "Module Name"
```

.IF

Conditional assembly block directive

Format

```
.IF  condition argument(s)
     .
     .
     .

range
     .
     .
     .

.ENDC
```

Parameters

condition

A specified condition that must be met if the block is to be included in the assembly. The condition must be separated from the argument by a comma, space, or tab. Table 5–4 lists the conditions that can be tested by the conditional assembly directives.

argument(s)

One or more symbolic arguments or expressions of the specified conditional test. If the argument is an expression, it cannot contain any undefined symbols. The assembler converts relocatable arguments to absolute arguments by discarding the relocatable portion of the expression and using only the offset from the beginning of the psect. Arguments must be separated by a comma.

range

The block of source code that is conditionally included in the assembly.

Table 5–4 Condition Tests for Conditional Assembly Directives

Condition Test		Complement Condition Test		Argument Type	Number of Arguments	Condition That Assembles Block
Long Form	Short Form	Long Form	Short Form			
EQUAL	EQ	NOT_EQUAL	NE	Expression	1 or 2[1]	Expression-1 is equal to expression-2 or not equal to expression-2.
GREATER	GT	LESS_EQUAL	LE	Expression	1 or 2[1]	Expression-1 is greater than expression-2 or less than or equal to expression-2.
LESS_THAN	LT	GREATER_EQUAL	GE	Expression	1 or 2[1]	Expression-1 is less than expression-2 or greater than or equal to expression-2.
DEFINED	DF	NOT_DEFINED	NDF	Symbolic	1	Symbol is defined or not defined.
BLANK	B	NOT_BLANK	NB	Macro	1	Argument is blank or not blank.
IDENTICAL	IDN	DIFFERENT	DIF	Macro	2	Arguments are identical or different.

[1]If the second argument is omitted, comparison is made against 0.

Description

A conditional assembly block is a series of source statements that are assembled only if a certain condition is met. A .IF starts the conditional block and a .ENDC ends the conditional block; each .IF must have a corresponding .ENDC. The .IF directive contains a condition test and one or two arguments. The condition test specified is applied to the arguments. If the test is met, all MACRO–64 statements between .IF and .ENDC are assembled. If the test is not met, the statements are not assembled. Optionally, you can use the .ELSE directive (or a combination of the .IFF, .IFT, and .IFTF directives) to specify an alternate series of statements to assemble if the test is not met.

You can nest conditional blocks; that is, a conditional block can be inside another conditional block. In this case, the statements in the inner conditional block are assembled only if the condition is met for both the outer and inner block.

Notes

- The assembler displays an error message if the following directives occur outside a conditional assembly block: .ENDC, .ELSE, .IF_FALSE, .IF_TRUE, .IF_TRUE_FALSE.
- MACRO–64 permits a nesting depth of 100 conditional assembly levels. If a statement attempts to exceed this nesting level depth, the assembler displays an error message.
- The effect of logical expressions can only be achieved by using several levels of .IF directives. See Example 5.
- Lowercase string arguments are converted to uppercase before being compared, unless the string is surrounded by double quotes or /NAMES=AS_IS is specified on the command line.

- The assembler displays an error message if .IF specifies any of the following: a condition test other than those in Table 5–4, an illegal argument, or a null argument specified in an .IF directive.

Examples

Example 1

Here is an example of a conditional assembly directive:

```
.IF EQUAL  ALPHA+1        ; Assemble block if ALPHA+1=0. Do
  .                       ;   not assemble if ALPHA+1 not=0
  .
  .
.ENDC
```

Example 2

Nested conditional directives take the following form:

```
.IF   condition argument(s)
.IF   condition argument(s)
  .
  .
  .
.ENDC
.ENDC
```

Example 3

The following conditional directives can govern whether assembly of the specified range is to occur:

```
.IF DEFINED  SYM1
.IF DEFINED  SYM2
  .
  .
  .
.ENDC
.ENDC
```

In this example, if the outermost condition is not satisfied, no deeper level of evaluation of nested conditional statements within the program occurs. Therefore, both SYM1 and SYM2 must be defined for the specified range to be assembled.

Example 4

An alternate series of statements can be specified using .ELSE.

```
.IF EQUAL A,B        ; Assemble if A is equal to B
.ELSE                ; Assemble if A is not equal to B
.ENDC
```

Example 5

The following example demonstrates the use of .ELSE and nesting:

```
.IF LESS_THAN X,Y  ; Assemble if X is less than Y
   .IF DEFINED Z   ; Assemble if Z is defined and
                   ;  X is less than Y
   .ELSE           ; Assemble if Z is not defined and
                   ;  X is less than Y
   .ENDC
.ELSE              ; Assemble if X is greater than or equal to Y
   .IF DEFINED Z   ; Assemble if Z is defined and X is
                   ;  greater than or equal to Y
   .ENDC
.ENDC
```

.IF_x

Subconditional assembly block directives

Format

.IF_FALSE

.IF_TRUE

.IF_TRUE_FALSE

Description

For compatibility with VAX MACRO, MACRO–64 provides three directives for use within .IF blocks:

Directive	Function
.IF_FALSE	If the condition of the assembly block tests false, the program includes the source code following the .IF_FALSE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE	If the condition of the assembly block tests true, the program includes the source code following the .IF_TRUE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE_FALSE	Regardless of whether the condition of the assembly block tests true or false, the source code following the .IF TRUE_FALSE directive (and continuing up to the next subconditional directive or to the end of the assembly block) is always included.

The implied argument of a subconditional directive is the condition test specified when the conditional assembly block was entered. A conditional or subconditional directive in a nested conditional assembly block is not evaluated if the preceding (or outer) condition in the block is not satisfied (see Example 3 and Example 4).

A conditional block with a subconditional directive is different from a nested conditional block. If the condition in the .IF is not met, the inner conditional blocks are not assembled, but a subconditional directive can cause a block to be assembled.

Notes

- If a subconditional directive appears outside a conditional assembly block, the assembler displays an error message.
- The alternate forms of .IF_FALSE, .IF_TRUE, and .IF_TRUE_FALSE are .IFF, .IFT, and .IFTF.
- You cannot use .ELSE in the same conditional block as .IF_x.

Examples

Example 1

Assume that symbol SYM is defined:

```
.IF DEFINED   SYM     ; Tests TRUE since SYM is defined.
  .                   ;   Assembles the following code.
  .
  .
.IF_FALSE             ; Tests FALSE since previous
  .                   ;   .IF was TRUE.  Does not
  .                   ;   assemble the following code.
  .
.IF_TRUE              ; Tests TRUE since SYM is defined.
  .                   ;   Assembles the following code.
  .
  .
.IF_TRUE_FALSE        ; Assembles following code
  .                   ;   unconditionally.
  .
  .
.IF_TRUE              ; Tests TRUE since SYM is defined.
  .                   ;   Assembles remainder of
  .                   ;   conditional assembly block.
  .
.ENDC
```

Example 2

Assume that symbol X is defined and that symbol Y is not defined:

```
.IF DEFINED  X        ; Tests TRUE since X is defined.
.IF DEFINED  Y        ; Tests FALSE since Y is not defined.
.IF_FALSE             ; Tests TRUE since Y is not defined.
  .                   ;   Assembles the following code.
  .
  .
.IF_TRUE              ; Tests FALSE since Y is not defined.
  .                   ;   Does not assemble the following
  .                   ;   code.
  .
.ENDC
.ENDC
```

Example 3

Assume that symbol A is defined and that symbol B is not defined:

```
.IF DEFINED  A       ; Tests TRUE since A is defined.
  .                  ;   Assembles the following code.
  .
  .
.IF_FALSE            ; Tests FALSE since A is defined.
  .                  ;   Does not assemble the following
  .                  ;   code.
  .
.IF NOT_DEFINED B    ; Nested conditional directive
  .                  ;   is not evaluated.
  .
  .
.ENDC
.ENDC
```

Example 4

Assume that symbol X is not defined but symbol Y is defined:

```
.IF DEFINED  X       ; Tests FALSE since X is not defined.
  .                  ;   Does not assemble the following
  .                  ;   code.
  .
.IF DEFINED  Y       ; Nested conditional directive
  .                  ;   is not evaluated.
  .
  .
.IF_FALSE            ; Nested subconditional
  .                  ;   directive is not evaluated.
  .
  .
.IF_TRUE             ; Nested subconditional
  .                  ;   directive is not evaluated.
  .
  .
.ENDC
.ENDC
```

.IIF

Immediate conditional assembly block directive

Format

.IIF condition [,]argument(s), statement

Parameters

condition

One of the legal condition tests defined for conditional assembly blocks in Table 5–4 (see the description of .IF). The condition must be separated from the arguments by a comma, space, or tab. If the first argument can be a blank, the condition must be separated from the arguments with a comma.

argument(s)

An expression or symbolic argument (described in Table 5–4) associated with the immediate conditional assembly block directive. If the argument is an expression, it cannot contain any undefined symbols. For more information on terms and expressions, see Section 2.8. The assembler converts relocatable arguments to absolute arguments by discarding the relocatable portion of the expression and using only the offset from the beginning of the psect. The arguments must be separated from the statement by a comma.

statement

The statement to be assembled if the condition is satisfied.

Description

.IIF provides a means of writing a one-line conditional assembly block. The condition to be tested and the conditional assembly block are expressed completely within the line containing the .IIF directive. No terminating .ENDC statement is required or allowed.

Notes

- The assembler displays an error message if .IIF specifies a condition test other than those listed in Table 5–4, which the assembler considers valid, an illegal argument, or a null argument.

Example

In the following example, the symbol EXAM is defined within the source program:

```
.IIF DEFINED EXAM, BR ALPHA
```

This directive generates the following code:

```
BR    ALPHA
```

.INCLUDE

Include source file directive

Format

.INCLUDE quoted-literal

Parameter

quoted-literal
The name of the source file to be included within double quotes. If a logical name exists that is the same as the source file name, specify the .M64 file extension to override the logical name. For more information on how to specify quoted literals, see Section 2.4.

Description

.INCLUDE indicates that the current input source file should be suspended and that the specified file should be used. When that file ends, the original source stream resumes, starting at the line after .INCLUDE.

Notes

- The assembler issues an error message if the file nesting level exceeds 50.

Example

```
.INCLUDE "file1.m64"
```

.INSTRUCTION

Instruction directive

Format

.INSTRUCTION expression

Parameter

expression
An absolute expression in the range of −2147483648 to 2147483647. For more information on terms and expressions, see Section 2.8. The expression cannot be relocatable, external, or complex.

Description

The specified value is stored at the current location as an instruction. You can use .INSTRUCTION to specify arbitrary instructions. Similar to .LONG, .INSTRUCTION stores a longword (32 bits) of data. Unlike .LONG, the assembler considers .INSTRUCTION an instruction and allows its use in code psects.

Notes

- You can only use this directive within code or mixed psects (psects that have either the EXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled within a mixed psect, this directive aligns the current location counter to a longword (32-bit) boundary before allocating storage.
- You can use this directive to store arbitrary, longword, assembly-time constant data in a code section.

Example

```
.INSTRUCTION 7
```

.IRP

Indefinite repeat argument directive

Format

```
.IRP   symbol,<argument list>
      .
      .
      .
range
      .
      .
      .
.ENDR
```

Parameters

symbol

A formal argument that is successively replaced with the specified actual arguments enclosed in angle brackets (<>). If no formal argument is specified, the assembler displays an error message.

<argument list>

A list of actual arguments enclosed in angle brackets and used in expanding the indefinite repeat range. An actual argument can consist of one or more characters. Multiple arguments must be separated by a legal separator (comma, space, or tab). If no actual arguments are specified, no action is taken.

range

The block of source text to be repeated once for each occurrence of an actual argument in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range and causes the current and remaining repetitions to be aborted.

Description

.IRP replaces a formal argument with successive actual arguments specified in an argument list. This replacement process occurs during the expansion of the indefinite repeat block range. The .ENDR directive specifies the end of the range.

.IRP is similar to a macro definition with only one formal argument. At each successive expansion of the repeat block, this formal argument is replaced with successive elements from the argument list. The directive and its range are coded in line within the source program. This type of macro definition and its range do not require calling the macro by name, as do other macros described in this section.

.IRP can appear either inside or outside another macro definition, indefinite repeat block, or repeat block (see the description of .REPEAT). The rules for specifying .IRP arguments are the same as those for specifying macro arguments.

Example

The macro definition is as follows:

```
.macro CHECK_PROCEDURE_KIND PROCEDURE_KIND
    OK = 0      ; Assume procedure_kind is unknown
    .irp REFERENCE_KIND,BOUND,NULL, REGISTER,STACK
        .if identical, <PROCEDURE_KIND>, <REFERENCE_KIND>
            OK = 1 ; Procedure_kind is known
            .mexit ; No need to look further
        .endc
    .endr
    .if eq, OK  ; If unknown procedure kind
        .error "Unknown procedure kind: PROCEDURE_KIND"
    .endc
.endm CHECK_PROCEDURE_KIND

CHECK_PROCEDURE_KIND REGISTER
CHECK_PROCEDURE_KIND FOOZLE
```

The macro call and expansion of the previously defined macro is as follows:

```
CHECK_PROCEDURE_KIND REGISTER
    OK = 0      ; Assume procedure kind is unknown
    .if identical,<REGISTER>,<BOUND>
    .endc
    .if identical,<REGISTER>,<NULL>
    .endc
    .if identical,<REGISTER>,<REGISTER>
        OK = 1 ; Procedure kind is known
        .mexit ; No need to look further
    .if eq, OK  ; If unknown procedure kind
    .endc

CHECK_PROCEDURE_KIND FOOZLE
    OK = 0      ; Assume procedure kind is unknown
    .if identical,<FOOZLE>,<BOUND>
    .endc
    .if identical,<FOOZLE>,<NULL>
    .endc
    .if identical,<FOOZLE>,<REGISTER>
    .endc
    .if identical,<FOOZLE>,<STACK>
    .endc
    .if eq, OK  ; If unknown procedure kind
        .error "Unknown procedure kind: FOOZLE"
    .endc
```

In this example the CHECK_PROCEDURE_KIND macro uses the .IRP directive to iterate over a list of reference keywords to determine if its argument matches one of the reference keywords. If a match is not found, the macro displays an error message.

.IRPC

Indefinite repeat character directive

Format

```
.IRPC   symbol,<STRING>
      .
      .
      .
range
      .
      .
      .
.ENDR
```

Parameters

symbol
A formal argument that is successively replaced with the specified characters enclosed in angle brackets (<>). If no formal argument is specified, the assembler displays an error message.

<STRING>
A sequence of characters enclosed in angle brackets and used in the expansion of the indefinite repeat range. Although the angle brackets are required only when the string contains separating characters, their use is recommended for legibility.

range
The block of source text to be repeated once for each occurrence of a character in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range.

Description

.IRPC is similar to .IRP except that .IRPC permits single-character substitution rather than argument substitution. On each iteration of the indefinite repeat range, the formal argument is replaced with each successive character in the specified string. The .ENDR directive specifies the end of the range.

.IRPC is similar to a macro definition with only one formal argument. At each expansion of the repeat block, this formal argument is replaced with successive characters from the actual argument string. The directive and its range are coded in line within the source program and do not require calling the macro by name.

.IRPC can appear either inside or outside another macro definition, indefinite repeat block, or repeat block (see description of .REPEAT).

Example

The macro definition is as follows:

```
.macro X_COUNT ARG
    COUNT = 0
    .irpc CH,<ARG>
        .iif identical,<CH>,<X>, COUNT = COUNT + 1
    .endr
.endm X_COUNT
```

The macro call and expansion of the macro defined previously is as follows:

```
X_COUNT XXFOOXBARXX
    COUNT = 0
    .irpc CH,<XXFOOXBARXX>
        .iif identical,<CH>,<X>, COUNT = COUNT + 1
    .endr
        .iif identical,<X>,<X>, COUNT = COUNT + 1
        .iif identical,<X>,<X>, COUNT = COUNT + 1
        .iif identical,<F>,<X>, COUNT = COUNT + 1
        .iif identical,<O>,<X>, COUNT = COUNT + 1
        .iif identical,<O>,<X>, COUNT = COUNT + 1
        .iif identical,<X>,<X>, COUNT = COUNT + 1
        .iif identical,<B>,<X>, COUNT = COUNT + 1
        .iif identical,<A>,<X>, COUNT = COUNT + 1
        .iif identical,<R>,<X>, COUNT = COUNT + 1
        .iif identical,<X>,<X>, COUNT = COUNT + 1
        .iif identical,<X>,<X>, COUNT = COUNT + 1

    .print "%integer(COUNT)"
%MACRO64-I-GENPRINT, Generated PRINT: 5
```

This example uses the .IRPC directive to iterate over the characters in the argument to the X_COUNT macro. Each time an argument character is X, the variable COUNT is incremented. After the X_COUNT macro has expanded, the example uses the %INTEGER( ) lexical operator to display the value of COUNT.

.LIBRARY

Macro library directive

Format

.LIBRARY quoted-literal1 [quoted-literal2]

Parameters

quoted-literal1

A string enclosed within double quotes that is the file specification of a macro library. If a logical name exists and it is the same as the macro library name, specify the .MLB file extension to override the logical name. For more information on quoted literals, see Section 2.4.

quoted-literal2

An optional string enclosed within double quotes that specifies a search list of file specifications where the assembler should look for the specified macro library. The assembler successively processes the search list in left-to-right order and attempts to locate the specified macro library in each location specified in the search list until the macro library is found. If the macro library is not found, the assembler issues a diagnostic message. If you omit the second argument to the .LIBRARY directive, the assembler uses the following search list by default:

- The current device and directory
- The device and directory specified by the logical name MACRO64$LIBRARY (if defined)
- The device and directory specified by the logical name ALPHA$LIBRARY (if defined)
- The device and directory specified by the logical name SYS$LIBRARY

Logical names may be defined as OpenVMS search lists.

Description

.LIBRARY adds a name to the macro library list that is searched whenever a .MCALL or an undefined opcode is encountered. The libraries are searched in the reverse order in which they were specified to the assembler.

If you omit any information from the macro-library-name file description, default values are assumed. The device defaults to your current default disk; the directory defaults to your current default directory; the file type defaults to MLB.

Example

```
.LIBRARY "MY_MACROS" ❶
.LIBRARY "PROJ_MACROS" "PROJ:[MACRO],PROJ:[DEVELOPMENT]" ❷
```

❶ The first statement adds the macro library MY_MACROS.MLB to the macro library list. The assembler first looks for MY_MACROS.MLB in the current directory. If not found there, the assembler next looks in the directory or directories indicated by the MACRO64$LIBRARY logical name if it is defined. If not found there or if MACRO64$LIBRARY is not defined, the assembler next looks in the directory or directories indicated by the ALPHA$LIBRARY

logical name if it is defined. If not found there or if ALPHA$LIBRARY is not defined, the assembler next looks in the directory or directories indicated by the SYS$LIBRARY logical name. If not found there, the assembler issues a diagnostic message.

❷ The second statement adds the macro library PROJ_MACROS.MLB to the macro library list. The assembler first looks for PROJ_MACROS.MLB in the PROJ:[MACRO] directory. If not found there, the assembler next looks in the PROJ:[DEVELOPMENT] directory. If not found there, the assembler issues a diagnostic message.

.LINKAGE_PAIR

Linkage directive

Format

.LINKAGE_PAIR name

Parameter

name
The name of the procedure descriptor, possibly defined in a different module of the routine to which linkage is required.

Description

.LINKAGE_PAIR causes a linkage pair to be stored at the current location counter. A linkage pair consists of two quadwords. The first quadword is the code entry point of the routine indicated by the specified identifier. (See .CODE_ADDRESS.) The second quadword is the address of the procedure descriptor of the routine indicated by the specified identifier. The second quadword is also called the procedure value. The specified name should reference a procedure descriptor that is either defined within the assembly unit or that becomes defined at the time the program is linked.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to an octaword (128-bit) boundary before allocating storage.

Example

```
.LINKAGE_PAIR A     ; Code address A followed by address of
                    ; procedure descriptor A
```

.LIST

Listing directive

Format

.LIST [argument-list]

Parameter

argument-list
One or more of the symbolic arguments defined in Table 5–7. (See the .SHOW directive for this table.) You can use either the long form or the short form of the arguments. If multiple arguments are specified, separate them with commas, spaces, or tabs.

Description

.LIST is equivalent to .SHOW. See the description of .SHOW for more information.

.LOCAL_CODE_ADDRESS

Local code address storage directive

Format

.LOCAL_CODE_ADDRESS name-list

Parameter

name-list
A list of symbols separated by commas. Each symbol references a procedure descriptor defined in the current module.

Description

.LOCAL_CODE_ADDRESS causes the code addresses of the specified identifiers to be placed at the current psect and current location counter. The specified identifier must reference a procedure descriptor defined within the module. The .LOCAL_CODE_ADDRESS directive, rather than the .CODE_ADDRESS directive, must be used with procedure descriptor names that are local (as opposed to global) to the assembly unit.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.

Example

```
.PROCEDURE_DESCRIPTOR P1,C1
.BLKQ 1
.LOCAL_CODE_ADDRESS P1       ; Code address
                             ; of P1...address of C1...
                             ; is stored here.
```

.LOCAL_LINKAGE_PAIR

Local linkage directive

Format

.LOCAL_LINKAGE_PAIR name

Parameter

name
The name of a procedure descriptor of the routine to which linkage is required. The specified procedure descriptor must be defined in the current module.

Description

.LOCAL_LINKAGE_PAIR causes a linkage pair to be stored at the current location counter. A linkage pair consists of a code address and the address of the specified identifier. The specified name must reference a procedure descriptor that is defined within the assembly unit.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to an octaword (128-bit) boundary before allocating storage.

Example

```
.PROCEDURE_DESCRIPTOR P1,CA1
 .
 .
 .
.LOCAL_LINKAGE_PAIR P1        ; Code address CA1 followed by
                              ; procedure descriptor address P1.
```

.LOCAL_PROCEDURE_DESCRIPTOR

Procedure descriptor labeling directive

Format

.LOCAL_PROCEDURE_DESCRIPTOR pd-name, ca-name

Parameters

pd-name
The name of the procedure descriptor. This name can be up to 31 characters long. It cannot be a temporary label.

ca-name
The name of the code address that corresponds to the procedure descriptor. This name must be defined later in the program as a label in a psect that has either the EXE or MIX attribute, or both. This name can be up to 31 characters long. It cannot be a temporary label.

Description

.LOCAL_PROCEDURE_DESCRIPTOR defines a bivalued local identifier that is used to represent a local routine. The first value is the procedure value, which is the address of the procedure descriptor. This value is defined as the current location counter at the point where you use the .LOCAL_PROCEDURE_DESCRIPTOR directive. The second value is the code address, which is the code entry-point address of the procedure. This value is defined by the second argument to the .LOCAL_PROCEDURE_DESCRIPTOR directive. No storage is allocated.

Notes

- See the *OpenVMS Calling Standard* for a full description of procedure descriptors.
- You must specify .LOCAL_PROCEDURE_DESCRIPTOR before the code of the routine it describes.
- See Section 6.7 for a description of the $PROCEDURE_DESCRIPTOR and $ROUTINE library macros. These macros define the procedure identifier and define the storage for the procedure descriptor.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information on psects, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before defining the procedure identifier.

Example

```
.LOCAL_PROCEDURE_DESCRIPTOR LP1,C1
```

.LONG

Longword storage directive

Format

.LONG expression-list

Parameter

expression-list
One or more expressions separated by commas.

Description

.LONG generates successive longwords (4 bytes) of data in the object module. The assembler truncates on the left of an integer or external value.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.3.
- If automatic data alignment is enabled, this directive aligns the current location counter to a longword (32-bit) boundary before allocating storage.
- You can define a 32-bit address item using macros and the .LONG directive. For example:

```
.macro address_32 item
.long item
.endm address_32
```

Example

```
.LONG 4     ; Places 4 in 4 bytes of storage.
```

.MACRO

Macro definition directive

Format

```
.MACRO   macro-name [formal-argument-list]
    .
    .
    .

range
    .
    .
    .

.ENDM   [macro-name]
```

Parameters

macro-name
The name of the macro to be defined; this name can be any legal symbol up to 31 characters long.

formal-argument-list
The symbols, separated by commas, to be replaced by the actual arguments in the macro call.

range
The source text to be included in the macro expansion.

Description

.MACRO begins the definition of a macro. It gives the macro name and a list of formal arguments. The .MACRO directive is followed by the source text to be included in the macro expansion. The .ENDM directive specifies the end of the range.

Macro names do not conflict with user-defined symbols. Both a macro and a user-defined symbol can have the same name.

When the assembler encounters a .MACRO directive, it adds the macro name to its macro name table and stores the source text of the macro (up to the matching .ENDM directive). No other processing occurs until the macro is expanded.

The symbols in the formal argument list are associated with the macro name and are limited to the scope of the definition of that macro. For this reason, the symbols that appear in the formal argument list can also appear elsewhere in the program.

For more information, see Section 4.1.

Notes

- If a macro has the same name as an Alpha AXP opcode, the macro is used instead of the instruction. This feature allows you to temporarily redefine an opcode.
- You can redefine a macro by using a .MACRO directive with the same name as in a previous macro definition. Therefore, the previous macro definition is implicitly deleted and the new macro definition supersedes the previous definition. See the .MDELETE directive for more information on macro deletion.
- You can nest a macro definition within another macro definition. The inner macro is not defined until the outer macro is invoked.
- You can nest a macro invocation so that one macro can invoke another. The assembler supports nested macro invocations to a depth of 1000. If a macro invokes itself, either directly or indirectly, it is *recursive*. Recursive macros must specify a basis-step in order to avoid infinite recursion. A basis-step is a macro exit condition that will eventually cause the macro to exit, which ends the recursion (see Example 3).

Examples

Example 1

This example shows how macro definitions and invocations may be nested. It also shows examples of the various forms of parameter passing. See Section 4.1 for more information on parameter passing.

```
.MACRO OP1 A,B=R4,?C
C:      ADDL R2, B, R3
.MACRO OP'B
        TRAPB
.ENDM OP'B
        ADDL A, R2, R3
        OP'B
        .MDELETE OP'B
.ENDM OP1
```

When OP1 is invoked "OP1 R0", the text expands to:

```
33000$: ADDL R2, R4, R3
.MACRO OPR4
        TRAPB
.ENDM
        ADDL R0, R2, R3
        OPR4
        .MDELETE OPR4
```

Processing this text will cause OPR4 to be expanded to TRAPB; the final text will be:

```
33000$: ADDL R2, R4 R3
        ADDL R0, R2, R3
        TRAPB
```

Example 2

The following example shows macro redefinition:

```
.MACRO INITIALIZE
   .MACRO INITIALIZE   ;Redefine to nothing
   .ENDM INITIALIZE
   X=0
   Y=1
   Z=-1
.ENDM INITIALIZE
```

Note that while the redefined version of the macro immediately supersedes the previous definition in any subsequent invocation, the invocation of the original definition expands to completion.

Example 3

This example shows a recursive macro:

```
.MACRO FACTORIAL N
   .IF EQUAL <N>,0      ;Basis step; stop at zero
      F=1
   .ELSE
      FACTORIAL <N-1>
      F = F * <N>
   .ENDC
.ENDM FACTORIAL
```

.MCALL

Macro call directive

Format

.MCALL macro-name-list

Parameter

macro-name-list
A list of macros to be defined for this assembly. Separate the macro names with commas.

Description

.MCALL specifies the names of the system and user-defined macros that are required to assemble the source program but are not defined in the source file.

If any named macro is not found upon completion of the search (that is, if the macro is not defined in any of the macro libraries), the assembler displays an error message.

Notes

- Using the .MCALL directive is optional unless the macro name is the same as an opcode or assembler directive. The assembler automatically searches for a library macro when it encounters an identifier that is not an opcode or directive in the opcode field. If your macro name is the same as an opcode or directive, you must use the .MCALL directive. You can also use the .MCALL directive in your program to document which macros are used by your program.

Example

```
.MCALL TRAPB       ; Substitute macro in library for
                   ; TRAPB instruction
```

.MDELETE

Macro deletion directive

Format

.MDELETE macro-name-list

Parameter

macro-name-list
A list of macros whose definitions are to be deleted. You can separate the macros with commas or spaces.

Description

.MDELETE deletes the definitions of specified macros.

.MDELETE completely deletes the macro. If you delete a macro that is currently expanding (such as a macro that deletes itself), the macro name is immediately removed from the macro name table and the macro is marked for deletion. When the macro finishes expanding, it is deleted.

Example

```
.MACRO FOO
.PRINT "In macro FOO"
.ENDM FOO
FOO
.MDELETE FOO
```

.MEXIT

Macro exit directive

Format

.MEXIT

Description

.MEXIT terminates a macro expansion before the end of the macro. Termination is the same as if .ENDM were encountered. You can also use the directive within repeat blocks. .MEXIT is useful in conditional expansion of macros and repeat blocks because it bypasses the complexities of nested conditional directives and alternate assembly paths.

Notes

- When .MEXIT occurs in a repeat block, the assembler terminates the current repetition of the range and suppresses further expansion of the repeat range.
- When macros or repeat blocks are nested, .MEXIT exits to the next higher level of expansion.
- If .MEXIT occurs outside a macro definition or a repeat block, the assembler displays an error message.

Examples

Example 1

The following macro definition uses the .MEXIT directive to exit the current macro when it has finished processing a particular kind of argument:

```
.macro STORE REG, LOCATION
    .if identical,<REG>,<FP>
        STQ REG, LOCATION
        .mexit
    .endc
    .if identical,<REG>,<SP>
        STQ REG, LOCATION
        .mexit
    .endc
    .if identical,<%extract(0,1,<REG>)>,<R>
        STQ REG, LOCATION
        .mexit
    .endc
    .if identical,<%extract(0,1,<REG>)>,<F>
        STT REG, LOCATION
        .mexit
    .endc
    .error "Register argument is not a register"
.endm STORE
```

Example 2

In this example, the STORE macro (as defined in Example 1) attempts to recognize its REG argument as either FP, SP, an integer register (a register with R as its first letter), or a floating-point register (a register with F as its first letter). The following example show two expansions of the STORE macro:

```
STORE R1, 0(SP)
     .if identical,<R1>,<FP>
     .endc
     .if identical,<R1>,<SP>
     .endc
     .if identical,<%extract(0,1,<R1>)>, <R>
     .if identical,<R>,<R>
         STQ R1, 0(SP)
         .mexit

 STORE 24(SP), 16(SP)
     .if identical,<24(SP)>,<FP>
     .endc
     .if identical,<24(SP)>,<SP>
     .endc
     .if identical,<%extract(0,1,<24(SP)>)>,<R>
     .if identical,<2>,<R>
     .endc
     .if identical, <%extract(0,1<24(SP)>)>,<F>
     .if identical,<2>,<F>
     .endc
     .error "Register argument is not a register"
```

The first call of the STORE macro stores R1 at 0(SP). The STORE macro determines to do an integer store by recognizing the letter R as the first letter of the register name. After it has done so, it abandons further expansion of the macro using the .MEXIT directive. The second invocation attempts to store 24(SP) at 16(SP). Since the STORE macro cannot identify 24(SP) as a legitimate register in any of the four forms it recognizes, the STORE macro does not attempt to store the REG argument, and does not abandon expansion with the .MEXIT directive. Instead, the STORE macro expands and issues a diagnostic message.

.NARG

Number of arguments directive

Format

.NARG symbol

Parameter

symbol
A symbol that is assigned a value equal to the number of positional arguments in the macro call.

Description

.NARG determines the number of arguments in the current macro call.

.NARG counts all the positional arguments specified in the macro call, including null arguments (specified by adjacent commas). The value assigned to the specified symbol does not include any keyword arguments or any formal arguments that have default values.

Notes

- If .NARG appears outside a macro, the assembler displays an error message.

Examples

Example 1

The macro definition is as follows:

```
.MACRO  CNT_ARG A1,A2,A3,A4,A5,A6,A7,A8,A9=DEF9,A10=DEF10
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
.ENDM   CNT_ARG
```

Example 2

The macro calls and expansions of the macro previously defined are as follows:

```
CNT_ARG TEST,FIND,ANS   ; COUNTER will = 3
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER

CNT_ARG                 ; COUNTER will = 0
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER

CNT_ARG TEST,A2=SYMB2,A3=SY3      ; COUNTER will = 1
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
                        ; Keyword arguments are not counted

CNT_ARG ,SYMBL,,        ; COUNTER will = 4
.NARG   COUNTER         ; COUNTER is set to no. of ARGS
.WORD   COUNTER         ; Store value of COUNTER
                        ; Null arguments are counted
```

.NCHR

Number of characters directive

Format

.NCHR symbol,<string>

Parameters

symbol

A symbol that is assigned a value equal to the number of characters in the specified character string.

<string>

A sequence of printable characters. Delimit the character string with angle brackets (<>) (or a character preceded by a circumflex (^)) only if the specified character string contains a legal separator (comma, space, or tab) or a semicolon (;).

Description

.NCHR determines the number of characters in a specified character string. It can appear anywhere in an MACRO–64 program and is useful in calculating the length of macro arguments.

Notes

- You can use the %LENGTH lexical operator instead of the .NCHR directive.

Examples

Example 1

The macro definition is as follows:

```
.MACRO   CHAR    MESS                  ; Define MACRO
.NCHR    CHRCNT,<MESS>      ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   "MESS"                        ; Store characters
.ENDM    CHAR                          ; Finish
```

Example 2

The macro calls and expansions of the macro previously defined are as follows:

```
CHAR     <HELLO>                        ; CHRCNT will = 5
.NCHR    CHRCNT,<HELLO>                 ; Assign value to CHRCNT
.WORD    CHRCNT                          ; Store value
.ASCII   "HELLO"                         ; Store characters

CHAR     <14, 75.39  4>                 ; CHRCNT will = 12(dec)
.NCHR    CHRCNT,<14, 75.39  4>          ; Assign value to CHRCNT
.WORD    CHRCNT                          ; Store value
.ASCII   "14, 75.39  4"                  ; Store characters
```

.NLIST

Listing exclusion directive

Format

.NLIST [argument-list]

Parameter

argument-list
One or more of the symbolic arguments listed in Table 5–7. (See the .SHOW directive for this table.) Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas, spaces, or tabs.

Description

.NLIST is equivalent to .NOSHOW. See the description of .SHOW for more information.

.NOSHOW

Listing exclusion directive

Format

.NOSHOW [argument-list]

Parameter

argument-list
One or more of the symbolic arguments listed in Table 5–7 in the description of .SHOW. Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas, spaces, or tabs.

Description

.NOSHOW specifies listing control options. See the description of .SHOW for more information.

.OCTA

Octaword storage directive

Format

.OCTA expression-list

Parameter

expression-list
A list of constant values separated by commas. Each value results in a 64-bit value being sign-extended to 128 bits and stored in an octaword.

Description

.OCTA generates 128 bits (16 bytes) of binary data.

Notes

- The low quadword contains the specified constant value.
- The high quadword contains the sign extension of the specified constant value. That is, the high quadword contains 0 if the specified value is positive, and it contains all bits set to 1 if the specified value is negative.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to an octaword (128-bit) boundary before allocating storage.

Example

```
.OCTA  0                        ; OCTA 0
.OCTA  ^X01234ABCD5678F9        ; OCTA hex value specified

.OCTA  VINTERVAL                ; VINTERVAL has 64-bit value,

                                ;   sign-extended
```

.ODD

Odd location counter alignment directive

Format

.ODD

Description

.ODD ensures that the current value of the location counter is odd by adding 1 if the current value is even. If the current value is already odd, no action is taken.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.3.

.PACKED

Packed decimal string storage directive

Format

.PACKED decimal-string[,symbol]

Description

.PACKED is supplied as a library macro with MACRO–64. For information about this directive, see the description of .PACKED in Chapter 6.

.PAGE

Page ejection directive

Format

.PAGE

Description

.PAGE forces a new page in the listing. The directive itself is also printed in the listing and begins the new page.

.PAGE in a macro definition is ignored. The paging operation is performed only during macro expansion. If the .PAGE directive occurs during macro expansion, text beginning with the original macro invocation line appears at the top of a new page.

Example

```
.MACRO SKIP    ; macro definition with .PAGE
.PAGE          ;
.ENDM SKIP     ;
.PSECT A,NOEXE ;
.BLKW 10       ;
SKIP           ; In the listing file, a form feed
               ;  will be inserted here
```

.PRINT

Assembly message directive

Format

.PRINT quoted-literal

Parameter

quoted-literal
The string of characters enclosed in quotes are displayed when encountered during assembly. For more information on quoted literals, see Section 2.4.

Description

.PRINT causes the assembler to display an informational message. The message consists of the string.

Notes

- .PRINT, .WARN, and .ERROR are directives that display messages. You can use these to display information indicating unexpected or important conditions within the assembly.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.PRINT "Questionable usage"
^
%MACRO64-I-GENPRINT, Generated PRINT: Questionable usage
at line number 3 in file DISK$:[TEST]PRINT.M64;2
```

.PROCEDURE_DESCRIPTOR

Procedure descriptor labeling directive

Format

.PROCEDURE_DESCRIPTOR pd-name, ca-name

Parameters

pd-name
The name of the procedure descriptor. This name can be up to 31 characters long. It cannot be a temporary label.

ca-name
The name of the code address that corresponds to the procedure descriptor. This name must be defined later in the program as a label in a psect that has either the EXE or MIX attribute, or both. This name can be up to 31 characters long. It cannot be a temporary label.

Description

.PROCEDURE_DESCRIPTOR defines a bivalued global identifier that is used to represent a global routine. The first value is the procedure value, which is the address of the procedure descriptor. This value is defined as the current location counter at the point where you use the .PROCEDURE_DESCRIPTOR directive. The second value is the code address, which is the code entry-point address of the procedure. This value is defined by the second argument to the .PROCEDURE_DESCRIPTOR directive. No storage is allocated.

Notes

- See the *OpenVMS Calling Standard* for a full description of procedure descriptors.
- You must specify .PROCEDURE_DESCRIPTOR before the code of the routine it describes.
- See Section 6.7 for a description of the $PROCEDURE_DESCRIPTOR and $ROUTINE library macros. These macros define the procedure identifier and define the storage for the procedure descriptor.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before defining the procedure identifier.

.PSECT

Program sectioning directive

Format

.PSECT program-section-name[,argument-list]

Parameters

program-section-name
The name of the program section (psect). For more information on psects, see Section 5.1.

argument-list
A list containing the program section attributes and the program section alignment. Table 5–5 lists the attributes and their functions. Table 5–6 lists the default attributes and their opposites. Program sections are aligned when you specify an integer in the range of 0 to 16 or one of the five keywords listed in the following table. If you specify an integer, the program section is linked to begin at the next virtual address that is a multiple of two raised to the power of the integer. If you specify a keyword, the program section is linked to begin at the next virtual address that is a multiple of the corresponding value listed in the following table:

Keyword	Size (in Bytes)
BYTE	$2^0 = 1$
WORD	$2^1 = 2$
LONG	$2^2 = 4$
QUAD	$2^3 = 8$
OCTA	$2^4 = 16$

QUAD is the default.

Table 5–5 Program Section Attributes

Attribute	Function
ABS	Absolute—The program section has an absolute address. An absolute program section contributes no binary code to the image, so its byte allocation request to the linker is 0. You cannot store initial values in an absolute program section with directives such as .BYTE, .WORD, .LONG, .QUAD. Usually the .BLKx directives are used in conjunction with label definitions within an absolute program section to define symbolic offsets within a structure. Compare this attribute with its opposite, REL.
CON	Concatenate—Program sections with the same name and attributes (including CON) from other modules are concatenated into one program section at link time. Their contents are concatenated in the order in which the linker acquires them. The allocated virtual address space is the sum of the individual requested allocations. Compare this attribute with its opposite, OVL.

(continued on next page)

Table 5–5 (Cont.) Program Section Attributes

Attribute	Function
EXE	Executable—The program section contains instructions. This attribute provides the capability of separating instructions from read-only and read/write data. The linker uses this attribute in gathering program sections and in verifying that the transfer address is in an executable program section. The assembler only allows you to place instructions in a program section that has either or both the EXE or MIX attributes. Compare this attribute with its opposite, NOEXE.
GBL	Global—Program sections that have the same name and attributes will be combined at link time into a single program section even when the individual program sections are in different clusters. This attribute is specified for Fortran COMMON block program sections. Compare this attribute with its opposite, LCL. For more information about clusters, see the *OpenVMS Linker Utility Manual*.
LCL	Local—The program section is restricted to its cluster. Compare this attribute with its opposite, GBL.
MIX	Mix—The program section can contain both data and instructions. The MIX and NOMIX attributes are assembly-time attributes that only affect assembler processing. The MIX and NOMIX attributes do not appear in the object module and do not affect linker processing. To mix instructions and data in the same psect, you must specify the MIX attribute. The NOMIX attribute is the default. If you choose to use instructions in a NOEXE psect, or use data directives in an EXE psect, you must specify the MIX attribute. The following limitations apply when using the MIX attribute: • Optimizations and alignment of code labels are not performed. Optimizations and code-label alignment are not performed on instructions placed in a MIX psect, regardless of whether the .ENABLE/.DISABLE options have been set, or if the command-line /OPTIMIZATION and /ALIGNMENT=CODE options have been specified. • Limited debugging information is provided. No PC-line (program counter) correlation information is generated for the instructions placed in a MIX psect. • The listing file includes only binary encoded instructions. All instructions that are placed in a MIX psect appear in the machine-code portion of the listing file, in their binary encoded instruction form as a data initializer to the .LONG data directive. There are no restrictions on data directives. Compare this attribute with its opposite, NOMIX. For more information, see Section 5.1.
NOEXE	Not Executable—The program section contains data only; it does not contain instructions. The assembler only allows you to place data allocations in a program section that has either or both the NOEXE and MIX attributes. Compare this attribute with its opposite, EXE.
NOMIX	The default attribute when you use the .PSECT directive. Compare this attribute with its opposite, MIX.
NOPIC	Non-Position-Independent Content—The program section is assigned to a fixed location in virtual memory (when it is in a shareable image). Compare this attribute with its opposite, PIC.

(continued on next page)

Table 5–5 (Cont.) Program Section Attributes

Attribute	Function
NORD	Nonreadable—Reserved for future use. Compare this attribute with its opposite, RD.
NOSHR	No Share—The program section is reserved for private use at execution time by the initiating process. Compare this attribute with its opposite, SHR.
NOWRT	Nonwritable—The contents of the program section cannot be altered (written into) at execution time. Compare this attribute with its opposite, WRT.
OVR	Overlay—Program sections with the same name and attributes (including OVR) from other modules receive the same relocatable base address in memory at link time. The allocated virtual address space is the requested allocation of the largest overlaying program section. Compare this attribute with its opposite, CON.
PIC	Position-Independent Content—The program section can be relocated; that is, it can be assigned to any memory area (when it is in a shareable image). Compare this attribute with its opposite, NOPIC. For more information about shareable images, see the *OpenVMS Linker Utility Manual*.
RD	Readable—Reserved for future use. Compare this attribute with its opposite, NORD.
REL	Relocatable—The linker assigns the program section a relocatable base address. The contents of the program section can be code or data. Compare this attribute with its opposite, ABS.
SHR	Share—The program section can be shared at execution time by multiple processes. This attribute is assigned to a program section that can be linked into a shareable image. Compare this attribute with its opposite, NOSHR.
WRT	Write—The contents of the program section can be altered (written into) at execution time. Compare this attribute with its opposite, NOWRT.

Table 5–6 Default Program Section Attributes

Default Attribute	Opposite Attribute
CON	OVR
EXE	NOEXE
LCL	GBL
NOMIX	MIX
NOPIC	PIC
NOSHR	SHR
RD	NORD
REL	ABS
WRT	NOWRT

Description

.PSECT defines a program section and its attributes and refers to a program section after it is defined. Use program sections to do the following:

- Develop modular programs.
- Separate instructions from data.
- Allow different modules to access the same data.
- Protect read-only data and instructions from being modified.

- Identify sections of the object module to the linker and the debugger.
- Control the order in which program sections are stored in virtual memory.

When the assembler encounters a .PSECT directive that specifies a new program section name, it creates a new program section and stores the name, attributes, and alignment of the program section. The assembler includes all data or instructions that follow the .PSECT directive in that program section until it encounters another .PSECT directive. The assembler starts all program sections at a relative location counter of 0.

The assembler does not automatically define program sections. Any code or data placed before the first .PSECT directive in the source code produces an assembly error.

If the assembler encounters a .PSECT directive that specifies the name of a previously defined program section, it stores the new data or instructions after the last entry in the previously defined program section, even with program sections that have the OVR attribute. (OVR program sections from separate modules are overlaid by the linker. The OVR attribute does not affect how multiple contributions to a psect are processed within a single assembly unit.) You need not relist the attributes when continuing a program section, but any attributes that are listed must be the same as those previously in effect for the program section. A continuation of a program section cannot contain attributes conflicting with those specified, or defaulted, in the original .PSECT directive.

The attributes listed in the .PSECT directive describe the contents of the program section. Except for the EXE and NOEXE attributes, the assembler does not check to ensure that the contents of the program section actually adhere to the attributes listed. However, the assembler and the linker do check that all program sections with the same name have exactly the same attributes. The assembler and linker display an error message if the program section attributes are not consistent.

Program section names are independent of local symbol, global symbol, and macro names. You can use the same symbolic name for a program section and for a local symbol, global symbol, or macro name. You may want to use unique names for clarity and maintainability.

Notes

- The .ALIGN directive cannot specify an alignment greater than that of the current program section; consequently, .PSECT should specify the largest alignment needed in the program section.
- For efficiency of execution and ease of programming, an alignment of quadword or larger is recommended for all program sections that have quadword data.

Example

```
.PSECT A,QUAD,EXE     ; Code psect
```

.QUAD

Quadword storage directive

Format

.QUAD expression-list

Parameter

expression-list
One or more expressions separated by commas.

Description

.QUAD generates 64 bits (8 bytes) of binary data.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.

Example

```
A:: .QUAD 4
```

.REPEAT
.REPT

Repeat block directive

Format

```
.REPEAT expression
    .
    .
    .

range
    .
    .
    .

.ENDR
```

Parameters

expression
An expression whose value controls the number of times the range is to be assembled within the program. When the expression is less than or equal to 0, the repeat block is not assembled. The expression must be absolute or relocatable and must not contain any undefined symbols. The assembler converts a relocatable value to the relative offset within the psect.

range
The source text to be repeated the number of times specified by the value of the expression. The repeat block can contain macro definitions or other repeat blocks. .MEXIT is legal within the range and causes the current and all succeeding repetitions to be aborted.

Description

.REPEAT repeats a block of code a specified number of times in line with other source code. The .ENDR directive specifies the end of the range.

Notes

The alternate form of .REPEAT is .REPT.

Examples

Example 1

The following macro definition uses the .REPEAT directive to store an ASCII string a specified number of times, followed by a 0 byte:

```
.MACRO  COPIES  STRING,NUM
.REPEAT NUM
.ASCII  "STRING"
.ENDR
.BYTE   0
.ENDM   COPIES
```

Example 2

The following macro call stores five copies of the string ABCDEF. This example is divided into four parts:

Macro invocation:

```
COPIES  <ABCDEF>,5
```

Macro expansion of .REPEAT invocation:

```
  .REPEAT 5
  .ASCII  "ABCDEF"
  .ENDR
```

.REPEAT expansion:

```
  .ASCII  "ABCDEF"
  .ASCII  "ABCDEF"
  .ASCII  "ABCDEF"
  .ASCII  "ABCDEF"
  .ASCII  "ABCDEF"
```

End of macro expansion:

```
  .BYTE   0
```

Example 3

The following macro call stores three copies of the string How Many Times. This example is divided into four parts:

Macro invocation:

```
  VARB = 3
  COPIES  <How Many Times>,VARB
```

Macro expansion of .REPEAT invocation:

```
  .REPEAT VARB
  .ASCII  "How Many Times"
  .ENDR
```

.REPEAT expansion:

```
  .ASCII  "How Many Times"
  .ASCII  "How Many Times"
  .ASCII  "How Many Times"
```

End of macro expansion:

```
  .BYTE   0
```

.RESTORE_PSECT
.RESTORE

Restore previous program section context directive

Format

.RESTORE_PSECT

.RESTORE

Description

.RESTORE_PSECT retrieves the program section from the top of the program section context stack, an internal stack in the assembler. If the stack is empty when .RESTORE_PSECT is issued, the assembler displays an error message. When .RESTORE_PSECT retrieves a program section, it restores the current location counter to the value it had when the program section was saved. The maximum stack level is 50. See the description of .SAVE_PSECT for more information.

Notes

- The alternate form of .RESTORE_PSECT is .RESTORE.
- You cannot use .RESTORE_PSECT to overwrite previous data-storage initializations. In the following example, MACRO–64 attempts to store 42 over 43 and fails, resulting in a diagnostic:

```
.PSECT A
.SAVE PSECT
.PSECT A
.QUAD 43
.RESTORE PSECT
.QUAD 42
```

Example

```
    .PSECT A,QUAD,NOEXE
A1: .WORD 5
A2: .QUAD 6
    .SAVE_PSECT        ; Saves psect A context
    .PSECT B,QUAD,NOEXE
B1: .WORD 6
    .RESTORE_PSECT     ; Return A location counter
A3: .WORD 5

    .PSECT B,QUAD,NOEXE
1$: .WORD 5
    .SAVE LOCAL_BLOCK  ; Saves psect B context and temporary
                       ; label context

    .PSECT C,NOEXE
1$: .WORD 6
    .RESTORE_PSECT     ; Restores psect B and saves
                       ; label context
    .ADDRESS 1$        ; References the address of
                       ; psect B temporary label 1$
```

.SAVE_PSECT
.SAVE

Save current program section context directive

Format

.SAVE_PSECT [LOCAL_BLOCK]

.SAVE [LOCAL_BLOCK]

Description

.SAVE_PSECT stores the current program section context on the top of the program section context stack, an internal assembler stack. It leaves the current program section context in effect. The program section context stack can hold up to 50 entries. Each entry includes the value of the current location counter and the maximum value assigned to the location counter in the current program section. If the stack is full when .SAVE_PSECT is encountered, an error occurs.

If the LOCAL_BLOCK option is specified, the current temporary label block is saved with the current program section context.

.SAVE_PSECT and .RESTORE_PSECT are especially useful in macros that define program sections. See the description of .RESTORE_PSECT for an example using .SAVE_PSECT.

Notes

- The alternate form of .SAVE_PSECT is .SAVE.

.S_FLOATING

Single-precision IEEE floating-point arithmetic directive

Format

.S_FLOATING floating-point-number-list

Parameter

floating-point-number-list
A list of IEEE single-precision floating-point constants separated by commas. For more information on floating-point numbers, see Section 2.3.2.

Description

.S_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .S_FLOATING generates 32-bit, single-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 23 bits of fractional significance). See the description of .T_FLOATING for information on storing double-precision floating-point IEEE numbers and the descriptions of .D_FLOATING, .F_FLOATING, and .G_FLOATING for descriptions of other floating-point numbers.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.3.
- If automatic data alignment is enabled, this directive aligns the current location counter to a longword (32-bit) boundary before allocating storage.

Example

```
.S_FLOATING 2.0,3.0,4.405
```

.SHOW
.NOSHOW

Listing inclusion and exclusion directives

Format

.SHOW [argument-list]

.NOSHOW [argument-list]

Parameter

[argument-list]
Use one or more of the symbolic arguments defined in Table 5–7. You can use either the long form or the short form of the arguments. If you specify multiple arguments, you must separate them by commas. If any argument is not specifically included in a listing control statement, the assembler assumes its default value (show or noshow) throughout the source program.

Table 5–7 .SHOW and .NOSHOW Symbolic Arguments

Long Form	Short Form	Default	Function
BINARY	MEB	Noshow	Lists macro and repeat block expansions that generate binary code. BINARY is a subset of EXPANSIONS.
CONDITIONALS	CND	Noshow	Lists unsatisfied conditional code associated with the conditional assembly directives.
EXPANSIONS	ME	Noshow	Lists macro and repeat range expansions.
LIBRARY	None	Noshow	Includes the macro definitions in a library in the listing.
INCLUDE	None	Noshow	Lists include file text in the listing file.

Description

.SHOW and .NOSHOW specify listing control options in the source text of a program. You can use .SHOW and .NOSHOW with or without an argument list.

When you use them with an argument list, .SHOW includes and .NOSHOW excludes the lines specified in Table 5–7. .SHOW and .NOSHOW control the listing of the source lines that are in conditional assembly blocks (see the description of .IF), macros, and repeat blocks. When you use them without arguments, these directives alter the listing level count. The listing level count is initialized to 0. Each time .SHOW appears in a program, the listing level count is incremented; Each time .NOSHOW appears in a program, the listing level count is decremented.

When the listing level count is negative, the listing is suppressed unless the line contains an error. Conversely, when the listing level count is positive, the listing is generated. When the count is 0, the line is either listed or suppressed, depending on the value of the listing control symbolic arguments.

Notes

- The listing level count allows macros to be listed selectively; a macro definition can specify .NOSHOW at the beginning to decrement the listing count and can specify .SHOW at the end to restore the listing count to its original value.
- The alternate forms of .SHOW and .NOSHOW are .LIST and .NLIST.
- The initial setting for each .LIST/.SHOW option (except BINARY) is obtained from the command-line setting using the /SHOW qualifier. For more information on command-line qualifiers, see Section 1.2.
- The /[NO]SHOW=BINARY option overrides the .[NO]SHOW BINARY directive.

Example

```
.NOSHOW   ; Turn off listing file display.  Counter < 0.
   .
   .
   .
.SHOW     ; Turn on listing file display.  Counter = 0.
          ; Value of .SHOW options are used.
   .
   .
   .
.SHOW     ; Counter > 0.  Listing file display is
          ; on for all options regardless of setting.
   .
   .
   .
```

.SIGNED_BYTE

Signed byte storage directive

Format

.SIGNED_BYTE expression-list

Parameters

expression-list
An expression or list of expressions separated by commas. Each expression specifies a value to be stored. The value must be in the range of −128 through +127.

Description

.SIGNED_BYTE generates successive bytes of binary data in the object module and performs signed range checking. Apart from the range check, .SIGNED_BYTE is equivalent to .BYTE for storage allocation.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.

Example

```
.PSECTA,NOEXE
.SIGNED_BYTE   LABEL1-LABEL2  ;  Data must fit
.SIGNED_BYTE   -126           ;     in a byte
```

.SIGNED_WORD

Signed word storage directive

Format

.SIGNED_WORD expression-list

Parameter

expression-list
An expression or list of expressions separated by commas. Each expression specifies a value to be stored. The value must be in the range of −32,768 through +32,767.

Description

.SIGNED_WORD generates successive words of binary data in the object module and performs signed range checking. Apart from the range check, .SIGNED_WORD is equivalent to .WORD in terms of storage allocation.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a word (16-bit) boundary before allocating storage.

Example

```
.PSECT $DATA,NOEXE
.SIGNED_WORD -32766;
.SIGNED_WORD 32769 ;causes assembly error
```

.SUBTITLE
.SBTTL

Listing subtitle directive

Format

.SUBTITLE quoted-literal

.SBTTL quoted-literal

Parameter

quoted-literal
An ASCII string enclosed in quotes from 1 to 31 characters long; excess characters are truncated. For more information on quoted literals, see Section 2.4.

Description

.SUBTITLE causes the assembler to print the line of text as the subtitle on the second line of each assembly listing page. This subtitle text is printed on each page until altered by a subsequent .SUBTITLE directive in the program.

Notes

- The alternate form of .SUBTITLE is .SBTTL.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.SUBTITLE "Terminal Display Routines"
```

.T_FLOATING

Double-precision IEEE floating-point arithmetic directive

Format

.T_FLOATING floating-point-number-list

Parameter

floating-point-number-list
A list of IEEE double-precision floating-point constants separated by commas. For more information on floating-point numbers, see Section 2.3.2.

Description

.T_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .T_FLOATING generates 64-bit, double-precision, floating-point data (1 bit of sign, 11 bits of exponent, and 52 bits of fractional significance). See the description of .S_FLOATING for information on storing single-precision floating-point IEEE numbers and the descriptions of .D_FLOATING, .F_FLOATING, and .G_FLOATING for descriptions of other floating-point numbers.

Notes

- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a quadword (64-bit) boundary before allocating storage.

Example

```
.T_FLOATING 4.5036,6.034
```

.TITLE

Listing title directive

Format

.TITLE module-name ["listing-title"]

Parameters

module-name
Either a quoted literal or an identifier that specifies the module's title. For more information on quoted literals, see Section 2.4.

"listing-title"
Optional quoted literal that specifies a title to appear within the first line of each listing output file. For more information on quoted literals, see Section 2.4.

Description

.TITLE assigns a name to the object module.

Notes

- The module name specified with .TITLE bears no relationship to the file specification of the object module, as specified in the MACRO–64 command line. The object module name appears in the linker load map and is also the module name that the debugger and librarian recognize.
- If .TITLE is not specified, MACRO–64 assigns the default name (.MAIN.) to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered establishes the name for the entire object module.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.TITLE "MAIN" "Main Entry Point"
```

.UNDEFINE_REG

Undefine register symbol directive

Format

.UNDEFINE_REG regsym

Parameter

regsym
A currently defined floating-point or integer register symbol.

Description

The register symbol that you specify as the argument to the .UNDEFINED_ REG directive is no longer a register symbol. Starting with the statement that follows the .UNDEFINE_REG directive, you can use the symbol as a MACRO–64 identifier.

Notes

- If you specify a MACRO–64 identifier that is not currently defined as a register symbol, the .UNDEFINE_REG directive has no effect.

Example

```
.DEFINE_IREG  X1    R5      ; X1 is integer register 5
.UNDEFINE_REG X1            ; X1 is an identifier again
.DEFINE_IREG  X1    7       ; X1 is integer register 7 - no
                             ;  redefinition and no warning

 $ROUTINE F0                 ; Error: F0 is a register and
                             ;  cannot be used as an
                             ;  identifier
 .UNDEFINE_REG F0            ; F0 is no longer a register
 $ROUTINE F0                 ; Ok now
```

.WARN

Warning directive

Format

.WARN quoted-literal

Parameter

quoted-literal
The string of characters enclosed in quotes are displayed during assembly. For more information on quoted literals, see Section 2.4.

Description

.WARN causes the assembler to display a warning message on the terminal or in the batch log file, and in the listing file (if there is one).

Notes

- .PRINT, .WARN, and .ERROR are directives that display messages. You can use them to display information indicating an unexpected or important assembly condition.
- This directive also accepts VAX MACRO syntax. See the *VAX MACRO and Instruction Set Reference Manual* for details.

Example

```
.WARN "Illegal parameter value; 0 assumed"
^
%MACRO64-W-GENWARN, Generated WARNING: Illegal parameter value; 0 assumed
at line number 3 in file DISK$:[TEST]WARN.M64;2
```

.WEAK

Weak symbol attribute directive

Format

.WEAK symbol-list

Parameter

symbol-list
A list of identifiers separated by commas.

Description

.WEAK specifies symbols that are either defined externally in another module or defined globally in the current module. .WEAK suppresses any object library search for the symbol.

When .WEAK specifies a symbol that is not defined in the current module, the symbol is externally defined. If the linker finds the symbol's definition in another module, it uses that definition. If the linker does not find an external definition, the symbol has a value of 0 and the linker does not report an error. The linker does not search a library for the symbol, but if a module brought in from a library for another reason contains the symbol definition, the linker uses it.

When .WEAK specifies a symbol that is defined in the current module, the symbol is considered to be globally defined. However, if this module is inserted in an object library, this symbol is not inserted in the library's symbol table. Consequently, searching the library at link time to resolve this symbol does not cause the module to be included.

Example

```
    .WEAK A,B,C
A:: .WORD 5     ; A and B are weak global definitions
B:: .QUAD 6
    .ADDRESS C  ; C is a weak external reference
```

.WORD

Word storage directive

Format

.WORD expression-list

Parameter

expression-list
One or more expressions separated by commas. For more information on terms and expressions, see Section 2.8.

Description

.WORD generates successive words (2 bytes) of data in the object module.

Notes

- The expression is first evaluated as a quadword and then truncated to a word. The value of the expression should be in the range of −32,768 to +32,767 for signed data or 0 to 65,535 for unsigned data. The assembler displays an error if the high-order 6 bytes of the quadword expression have a value other than zero or ^XFFFFFFFFFFFF.
- The assembler truncates on the left of an integer or external value.
- Addresses are not allowed with .WORD.
- You can only use this directive within data or mixed psects (psects that have either the NOEXE or MIX attributes). For more information, see Section 5.1.
- If automatic data alignment is enabled, this directive aligns the current location counter to a word (16-bit) boundary before allocating storage.

Example

```
.WORD 5,6,7
```

6
MACRO–64 Supplied Macros

This chapter describes the library macros supplied with the MACRO–64 assembler. These macros make it easy for you to define and call routines. For a detailed description of each macro, see Section 6.7.

This chapter also describes the $OPDEF macro, which allows you to define your own opcodes.

6.1 MACRO–64 Supplied Library

The MACRO–64 assembler provides a library of macros that help you to program in conformance with the OpenVMS Calling Standard. This library also allows you to define your own opcodes and packed decimal data. The library, called MACRO64.MLB, is installed on your system with the MACRO–64 assembler.

The MACRO–64 assembler automatically adds the MACRO64.MLB library to the library list before it begins processing your source program. Consequently, you do not need to add it manually using the .LIBRARY directive.

The following macros are supplied by the MACRO–64 Assembler:

$BEGIN_EPILOGUE
$CALL
$CODE_SECTION
$DATA_SECTION
$END_EPILOGUE
$END_PROLOGUE
$END_ROUTINE
$LINKAGE_PAIR
$LINKAGE_SECTION
$OPDEF
.PACKED
$PROCEDURE_DESCRIPTOR
$RESET_LP_LIST
$RETURN
$ROUTINE

Appendix B explains how to program more effectively using the MACRO–64 assembly language within the framework of the OpenVMS Calling Standard.

6.2 Routines and Lexical Scope

The calling-standard macros use the concept of a single, current, active routine. A routine is a programming entity that is associated with a procedure descriptor that may be called, or is a main routine specified as the transfer address of a linked image.

Only one routine can be active or current at any given time during assembly. If more than one routine is defined in a single assembler source file, all items associated with the current routine, that is, within the lexical scope of the routine, must be completed before making a different routine current. The lexical scope of one routine cannot overlap the lexical scope of another routine.

A routine becomes current or comes into scope by invoking the $ROUTINE macro with the appropriate arguments. $ROUTINE marks the beginning of the lexical scope of a routine. The complementary macro, $END_ROUTINE, marks the end of the current routine's lexical scope.

6.2.1 Routines and Program Sections

Routines have three types of associated program sections:

- Code section—Contains the executable instructions of the routine. This section is typically read-only and executable.
- Data section—Contains data accessed by a routine. Typically, this is where variable data is stored. This section is typically nonexecutable, readable, and writeable.
- Linkage section—Contains a routine's procedure descriptor and the necessary linkage information for calling other routines, and for linkage to data not in the linkage section, if any. Also, constant data may be placed here. Typically, this section is read-only and not executable.

  The linkage section is considered a type of data section with the following function:

  - Provides linkage information for calls made from a routine associated with the linkage section.
  - Provides linkage information for data outside of the linkage section.
  - Defines the associated routine's procedure descriptor so that calls can be made to the routine.
  - Defines constant or static data.

6.3 Using Macros to Control Program Sections

The following section explains how to use the MACRO–64 supplied macros to define and control the various program sections during assembly.

6.3.1 Defining Program Sections

Each of the three program section types can be referenced by a corresponding macro. The following three macros are used to refer to and define these three psects associated within a routine:

- $CODE_SECTION—makes the routine's code psect current.
- $DATA_SECTION—makes the routine's data psect current.
- $LINKAGE_SECTION—makes the routine's linkage psect current.

To switch to one of the current routine's sections, you invoke the corresponding macro. For example, after invoking $ROUTINE, to place instructions into the current routine's code psect, you invoke the $CODE_SECTION macro. This makes the code psect associated with the current routine the current psect. Similarly, invoking $LINKAGE_SECTION makes the linkage psect associated with the current routine the current psect.

You can also control the psect name and attributes for each of the program sections by defining arguments to the $ROUTINE macro. For more information, see Section 6.7.

6.3.2 Using Macro-Defined Symbols

When you use any of the supplied macros described in this chapter, with the exception of $OPDEF, the following register symbols are defined for you:

```
.DEFINE_IREG $IA0 R16 ; Integer argument 0
.DEFINE_IREG $IA1 R17 ; Integer argument 1
.DEFINE_IREG $IA2 R18 ; Integer argument 2
.DEFINE_IREG $IA3 R19 ; Integer argument 3
.DEFINE_IREG $IA4 R20 ; Integer argument 4
.DEFINE_IREG $IA5 R21 ; Integer argument 5
.DEFINE_FREG $FA0 F16 ; Floating-point argument 0
.DEFINE_FREG $FA1 F17 ; Floating-point argument 1
.DEFINE_FREG $FA2 F18 ; Floating-point argument 2
.DEFINE_FREG $FA3 F19 ; Floating-point argument 3
.DEFINE_FREG $FA4 F20 ; Floating-point argument 4
.DEFINE_FREG $FA5 F21 ; Floating-point argument 5
.DEFINE_IREG $AI R25 ; Argument-information register
.DEFINE_IREG $RA R26 ; Return-address register
.DEFINE_IREG $PV R27 ; Procedure-value register
.DEFINE_IREG $FP R29 ; Frame pointer
.DEFINE_IREG $SP R30 ; Stack pointer
```

Note

SP and FP remain predefined by MACRO–64 whether or not you use the OpenVMS Calling Standard macros. $SP and $FP are defined by the OpenVMS Calling Standard macros for consistency with the other register definitions that correspond to OpenVMS Calling Standard register conventions.

The following symbols are defined by the $ROUTINE macro. These symbols are useful while programming with the calling-standard macros to refer to particular data and linkage section items.

- $CS—The address of the beginning of the current routine's code section.
- $LS—The address of the beginning of the current routine's linkage section.
- $DS—The address of the beginning of the current routine's data section.
- $DP—The linkage section address where the $ROUTINE macro places the .ADDRESS $DS to enable access to the data section from the linkage section (this variable is optional).
- $SIZE—The size of the fixed-stack area in bytes. $SIZE is defined using the value specified with the SIZE argument.

- $RSA_OFFSET—The offset within the fixed-stack area to the register save area. $RSA_OFFSET is defined using the value specified with the RSA_OFFSET argument.
- $RSA_END—The offset within the fixed-stack area to the first byte beyond the end of the register save area.

For more information, see the description for the $ROUTINE macro in this chapter.

The $CALL macro also defines the $STACK_ARG_SIZE symbol. This symbol specifies the number of bytes used to store arguments on the stack. This value is useful after calling a routine that returns a value on the stack. In this case, $CALL cannot remove the arguments from the stack because you must first retrieve the returned value from the stack. Subsequently, you can remove the arguments from the stack using the $STACK_ARG_SIZE symbol. For more information, see the description of $CALL in this chapter.

6.3.3 Defining Procedure Type

The OpenVMS Calling Standard defines four types of routines; stack, register, null, and bound. For more information on types of routines, see Section B.5 and the *OpenVMS Calling Standard*.

You can define the routine type by using the KIND keyword argument with $ROUTINE or $PROCEDURE_DESCRIPTOR macros. The validity and values of other $ROUTINE and $PROCEDURE_DESCRIPTOR macro parameters are determined by the type of routine being declared. For example, a null procedure type has no stack size; therefore, the SIZE parameter is invalid and cannot be specified for a null procedure type. When using the KIND keyword with the $ROUTINE or $PROCEDURE_DESCRIPTOR macros, note the following exceptions:

- The $SIZE symbol defined by $ROUTINE is only valid for stack and register routines.
- The $RSA_OFFSET symbol defined by $ROUTINE is only valid for a stack routine.

6.3.4 Using Macros in Prologue Sections

With stack and register routines, $ROUTINE generates a standard prologue sequence if you specify the STANDARD_PROLOGUE=TRUE keyword argument. STANDARD_PROLOGUE=TRUE is the default for register and stack routines.

Alternatively, you can code your own prologue sequence by specifying the STANDARD_PROLOGUE argument as FALSE and using the $END_PROLOGUE macro to mark the end of your prologue sequence. In this case, you may wish to use the $SIZE and $RSA_OFFSET symbols to symbolically specify the fixed-stack size and register save area offset, respectively.

For more information on the prologue sequence of instructions that must occur at the beginning of all stack and register routines, see the *OpenVMS Calling Standard*.

6.3.5 Using Macros in Epilogue Sections

With stack and register routines, you can use the $RETURN macro to generate an epilogue sequence. Alternatively, you can code your own epilogue sequence using the $BEGIN_EPILOGUE and $END_EPILOGUE macros to mark the beginning and end of your epilogue sequences.

The OpenVMS Calling Standard also describes the epilogue sequence of instructions that must be executed every time a stack or register routine returns to its caller.

6.4 Programming Examples Using Supplied Macros

Examples 6–1 and 6–2 show how to use the calling-standard macros to define a routine, switch control between psects, generate an epilogue sequence, and end the routine.

Example 6–1 Program Using Supplied Macros

```
        $ROUTINE MAIN, KIND=STACK, - ; Stack routine kind ❶
            SAVED_REGS=<FP>, - ; Saves FP          ❷
            SIZE=48             ; Stack size is 48  ❸

        $LINKAGE_SECTION        ; Switch to the linkage psect. ❹

X:      .long 6                 ; X is a constant
FP1_ADDR:                       ; FP1_ADDR -> FP1
        .address FP1

        $DATA_SECTION           ; Switch to the data section ❺

A::     .blkw 5                 ; $DS points here
B::     .blkw

        $CODE_SECTION           ; Switch to the code section ❻
                                ; ($CS points here)
                .
                .
                .
        $RETURN                 ; Perform epilogue and return ❼

        $END_ROUTINE MAIN       ; Mark the end of the routine ❽
```

❶ $ROUTINE defines the routine MAIN. The routine type is defined as a stack routine using the KIND=STACK keyword argument.

❷ The keyword argument SAVED_REGS=<FP> adds the frame pointer register to the list of registers saved on the stack by the prologue code. The SAVED_REGS keyword is valid only for stack procedures. If you do not specify the FP register (R29) with this argument, it is assumed.

❸ The keyword argument SIZE=48 defines the stack area in bytes. This argument is valid only for register and stack routines. If you do not specify a stack size for a stack or register routine, $ROUTINE computes the minimum stack size required to accommodate the other arguments you specify or leave as the default.

❹ The $LINKAGE_SECTION macro switches to the linkage section. You can use the $LS symbol created by the $ROUTINE macro to point to the address of the current routine's linkage section. $ROUTINE creates the procedure descriptor and leaves the current location counter within the linkage section just beyond the procedure descriptor.

❺ The $DATA_SECTION macro switches to the data section. You can use the $DS symbol created by the $ROUTINE macro to point to the address of the current routine's data section.

❻ The $CODE_SECTION macro switches to the code section. The $CS symbol created by the $ROUTINE macro is used to point to the address of the current routine's code section.

❼ The $RETURN macro generates a standard epilogue instruction sequence. You can use this macro only with stack or register routine defined with the $ROUTINE macro.

❽ The $END_ROUTINE macro marks the end of the routine.

6.5 Using the $CALL Macro

$CALL calls local or external routines previously defined by the $ROUTINE macro or defined in another language.

To call a routine using standard linkage, use the $CALL macro. You invoke this macro from within the routine's code section. $CALL performs the following actions:

- Searches a list of linkage pairs referenced in previous invocations of the $CALL and $LINKAGE_PAIR macros within the calling routine. If a linkage pair is already found to exist on the list, $CALL uses the linkage pair stored from the previous invocation. Otherwise, $CALL stores the linkage pair of the called routine in the caller's linkage section and adds the linkage pair to the caller's list.
- Allocates stack space for arguments, if necessary.
- Loads arguments specified with the ARGS argument into argument registers and onto the stack, as appropriate.
- Sets the arguments information register, R25, according to the arguments specified with ARGS argument.
- Generates the following instruction sequence to perform the actual call based on the location of the linkage pair generated from above, and the value of Rls, linkage register, which is assumed to point to the base of the linkage section:

```
LDQ R26, code_addr_offset(Rls)      ; load code address
LDQ R27, proc_desc_addr_offset(Rls) ; load procedure
                                    ; descriptor address
                                    ;
JSR R26, R26                        ; Jump to the routine
                                    ;  saving the return
                                    ;  address in R26
```

- Frees argument stack space, if any, and if the called routine does not return a value on the stack.

  Like $ROUTINE, the $CALL macro invokes other macros to perform the previous tasks.

  If you do not specify the Rls argument in your invocation of $CALL, $CALL assumes that you have used the .BASE directive to define a register that points to the base address of your linkage section. That is, it assumes that you have included a statement similar to the following:

```
.BASE R27, $LS
```

This source statement defines the base address of the linkage section to be R27, and to be associated with the macro symbol $LS. This source statement should be placed in your source code before the $CALL macro call.

6.5.1 Using $CALL in Source Code

Example 6–2 uses the source code from Example 6–1, except it uses the $CALL macro to show how to call a local and external routine.

Example 6–2 Program Using $CALL

```
    .
    .
    .
$CODE_SECTION             ; Switch to the code section
                          ; ($CS points here)
    .
    .
    .
MOV     R27,R2     ❶
.base   R2, $LS    ❷
    .
    .
    .
$CALL SUB1                ; Call external routine SUB1  ❸
$CALL SUB2, LOCAL=TRUE    ; Call local routine SUB2     ❹
    .
    .
    .
$RETURN                   ; Perform epilogue and return
$END_ROUTINE MAIN         ; Mark the end of the routine
```

❶ The $LS symbol is defined to be the address of the procedure that is defined within the linkage section. The calling routine places the address of the procedure descriptor in R27 before making the call. This guarantees that the address associated with the symbol $LS is stored in R27 upon routine entry. Since the information in R27 is erased during a standard call, a copy is preserved in register R2.

❷ Register R2 now contains the address of our procedure descriptor, which is the base address of our linkage section. Since $LS is defined to point to the base address of the linkage section, the assembler computes the offsets within the linkage section using the .BASE directive.

❸ The $CALL macro calls the external routine SUB1.

❹ The $CALL macro calls the local routine SUB2. This call uses the LOCAL=TRUE keyword argument to indicate that routine SUB2 is defined within the module.

6.6 Programming Considerations

This section discusses some programming considerations you need to be aware of when using the calling-standard macros.

6.6.1 Making Multiple Calls From the Same Routine

The $CALL macro generates the following instruction sequence:

```
LDQ R26, code_address_offset(Rls)              ; load code address
LDQ R27, procedure_descriptor_address_offset(Rls) ; load procedure
                                                ;    descriptor
                                                ;    address
JSR R26, R26                                    ;
```

The contents of R26 and R27 are erased as a result of using the $CALL macro. This is important since Rls in the previous sequence is typically R27. Thus, if you require subsequent access to your linkage section, such as when making subsequent calls, you need to make a working copy of R27 to use after the first call. Example 6–2 shows how this technique is used.

Note that $CALL also overwrites the values in the argument registers, and the scratch registers specified or the default set by the SCRATCH_REGS argument, when you pass arguments to the called routine.

6.6.2 Nonstandard Linkage

Under certain circumstances, there may be advantages in using a nonstandard routine linkage. For more information, see Appendix B and the *OpenVMS Calling Standard*.

6.6.3 Routine Restrictions

Different routine types have different capabilities and restrictions. For example, only a stack routine that specifies BASE_REG_IS_FP=TRUE can make standard calls. For more information, see the description for $ROUTINE in this chapter.

For a description of the different capabilities and restrictions associated with each routine type, see Section B.5 and the *OpenVMS Calling Standard*.

6.7 Macro Descriptions and Syntax Rules

This section provides a detailed description of the MACRO–64 macros as well as the syntax rules.

You can use either positional or keyword argument association or a combination of the two with these macros. For positional association, the order of formal arguments is shown with the format of each macro. For more information on argument association, see Chapter 4. The following syntax rules apply when invoking the assembler using the command-line qualifier /NAMES=AS_IS:

- When specifying macro names, you must use all uppercase or all lowercase characters. You cannot mix uppercase and lowercase characters.
- When specifying keyword arguments, you must use the same alphabetic case as the macro name it is associated with. If you use lowercase characters with the macro name, you must use lowercase characters with the keyword argument. If you use uppercase characters with the macro name, you must use uppercase characters with the keyword argument.

$BEGIN_EPILOGUE

Marks the beginning of an epilogue instruction sequence.

Format

$BEGIN_EPILOGUE

Description

$BEGIN_EPILOGUE marks the beginning of an epilogue instruction sequence that you code within a stack or register routine defined with the $ROUTINE macro.

At each point where a stack or register routine returns to its caller, the routine must perform a sequence of operations to restore any saved registers and to perform stack frame management. This sequence is called the epilogue and is described in detail in the *OpenVMS Calling Standard*.

You can use the $RETURN macro to generate a standard epilogue instruction sequence for you, or you can code your own sequence. If you code your own epilogue sequence, you must mark the beginning and end of the epilogue sequence with the $BEGIN_EPILOGUE and $END_EPILOGUE macros.

Notes

- You must not use $BEGIN_EPILOGUE for an epilogue instruction sequence generated by $RETURN. $RETURN automatically invokes $BEGIN_EPILOGUE and $END_EPILOGUE.

Example

```
$ROUTINE MUMBLE, KIND=REGISTER, SAVE_FP=R1
    :
    :
    :
$BEGIN_EPILOGUE
MOV     R1,FP               ; Restore caller's frame
RET     (R26)               ; Return to caller
$END_EPILOGUE
    :
    :
    :
$END_ROUTINE MUMBLE
```

$CALL

Issues a call to another routine.

Format

```
$CALL  NAME=routine-being-called -
       [Rls=linkage-section-register] -
       [LS=linkage-section-address] -
       [LOCAL=boolean] -
       [ARGS=argument-list] -
       [SET_ARG_INFO=boolean-value] -
       [STACK_RETURN_VALUE=boolean-value] -
       [SCRATCH_REGS=scratch_reg-list] -
       [TIE=boolean-value] -
       [FUNC_RETURN= {I64,D64,I32,U32,FF,FD,FG,FS,FT,FDC,FGC,FSC,FTC}] -
       [USES_VAX_ARGLIST=boolean-value] -
       [SIGNATURE_BLOCK=signature_address] -
       [NONSTANDARD=boolean-value] -
```

Parameters

NAME

The name of the routine to call. This argument is required.

Rls

Linkage section register to use when generating the LDQ, LDQ, JSR instruction sequence. This argument is optional.

If Rls is omitted, $CALL assumes that you entered a .BASE directive before invoking $CALL that establishes the value of a base register pointing into the linkage section. If you omit the Rls argument and you do not enter such a .BASE directive before invoking $CALL, the assembler issues the following error message during the $CALL macro expansion:

```
"Argument 2 invalid"  The assembler failed to find a
 base register specified with a previous .BASE directive
 to form a register expression of the form offset(Rn)"
```

LS

LS is the address of the linkage section. If you use $CALL within a routine defined by the $ROUTINE macro, the LS argument defaults to the $LS symbol defined by $ROUTINE. If you use $CALL outside of a routine defined by the $ROUTINE macro, there are two ways that you can indicate the location of the linkage section to $CALL. First, you can specify the LS argument to $CALL as a relocatable address expression that indicates the base of the linkage section. In this case you must also specify the Rls argument. Second, you can specify both the linkage-section base register and the linkage-section address in a .BASE directive before invoking $CALL. In this case, you must omit both the Rls and LS arguments to $CALL.

Digital recommends that you omit this argument if you use $CALL within a routine defined by the $ROUTINE macro.

LOCAL

A Boolean value (TRUE or FALSE) that specifies whether the routine to be called is local to the module or globally visible. By default, $CALL generates a call to a globally visible routine. To generate a call to a routine that is not globally visible, you must specify LOCAL=TRUE.

ARGS

An optional list of arguments to pass to the called routine. Enclose the argument list within angle brackets (<>) and separate the arguments with commas (,). You can use the following qualifiers with each argument you specify with the ARGS argument. Table 6–1 describes these qualifiers.

Each argument is an address expression (which may include a register) followed by a qualifier. The table also contains the argument type, the instruction used to load the argument into a register, the instruction used to store the argument on the stack, and the encodings used in the Argument Information Register (R25) in the call signature block when you specify TIE=TRUE. See the *OpenVMS Calling Standard* for more information on these encodings. Note that arguments are only stored on the stack if there are more than six arguments provided to the routine.

Table 6–1 ARGS Arguments

Argument Qualifier	Argument Type	LOAD Instruction	STORE Instruction	AI Encoding	Reg Arg Signature	Mem Arg Signature
/A	Address	LDA	STQ	I64	I32	I32
/D	D-floating	LDG	STG	FD	FD	Q
/F	F-floating	LDF	STF	FF	FF	I32
/G	G-floating	LDG	STG	FG	FG	Q
/L	Longword	LDL	STQ	I64	I32	I32
/Q	Quadword	LDQ	STQ	I64	Q	Q
/S	S-floating	LDS	STS	FS	FS	I32
/T	T-floating	LDT	STT	FT	FT	Q
/UL[1]	Unsigned Longword	LDL/ ZAP #^xF0	STQ	I64	U32	I32

[1]Unsigned 32-bit integers are normally passed using the /L argument qualifier. Therefore, Digital does not recommend that you use the /UL argument qualifier.

SET_ARG_INFO

An optional argument to indicate whether $CALL should set the Argument Information (AI) register (R25) with the appropriate argument information or not. By default, or if you specify SET_ARG_INFO=TRUE, $CALL stores the appropriate argument information in R25. If you specify SET_ARG_INFO=FALSE, $CALL does not affect R25.

If you want to conform to the OpenVMS Calling Standard, you must store the appropriate information in R25 yourself before invoking $CALL. If you do not need to conform to the OpenVMS Calling Standard, and if the called routine does not need argument information in R25, you can specify SET_ARG_INFO=FALSE and make no change in R25. By making no change in R25, you avoid the

overhead involved when either you or $CALL load argument information into R25 at the expense of calling standard conformance.

STACK_RETURN_VALUE

An optional argument to indicate that the called routine returns a value on the stack. By default, $CALL assumes that the called routine does not return a value on the stack. In this case, $CALL removes any arguments passed to the called routine from the stack when the called routine returns.

If the called routine returns a value on the stack, the returned value is placed at a lower address than the arguments on the stack. In this case, you must specify STACK_RETURN_VALUE=TRUE to prevent $CALL from removing the arguments to the called routine from the stack and erasing the value returned by the called routine. You must retrieve the return value and remove it from the stack. Then you can remove the arguments to the called routine using the $STACK_ARG_SIZE symbol defined by $CALL.

SCRATCH_REGS

An optional list of scratch registers for $CALL to use when processing arguments passed to the called routine with the ARGS argument. If you pass more than six arguments to the called routine, $CALL may need to use scratch registers to process the call.

By default, $CALL uses R0, R1, F0, and F1. You can cause $CALL to use different scratch registers with the SCRATCH_REGS argument.

If you are passing integer arguments, you should specify at least one integer register. If you are passing floating-point arguments, you should specify at least one floating-point register.

$CALL can process arguments to the called routine more efficiently if you specify two or more scratch registers of the type or types appropriate to the arguments you are passsing.

TIE

A Boolean value (TRUE or FALSE) that specifies whether $CALL should generate a call sequence that is compatible with both native routines and the Translated Image Environment (TIE). By default, $CALL generates a faster call sequence that is only compatible with native routines. If you specify TIE=TRUE, $CALL generates a call sequence that works with both native routines and translated routines. If you are calling a VAX routine in a shareable image that has been translated with the DECmigrate image translator, specify TIE=TRUE. If you are calling a native routine, Digital recommends you default the TIE argument or specify TIE=FALSE. While $CALL generates a call sequence that is compatible with native routines when you specify TIE=TRUE, that call sequence is slower than when you specify or default TIE=FALSE.

FUNC_RETURN

An optional argument used to indicate the type of function return when you also specify TIE=TRUE. This argument is ignored unless you also specify TIE=TRUE. Specify one of I64, D64, I32, U32, FF, FD, FG, FS, FT, FFC, FDC, FGC, FSC, or FTC. These values correspond to the RASE$K_FR_* signature encodings described in Table 3-7 in the *OpenVMS Calling Standard*. If you specify TIE=TRUE and do not specify a function return type with FUNC_RETURN, the default function return type is I64.

Note

Specification of the FUNC_RETURN argument does not in itself cause $ROUTINE to generate a procedure signature block. However, if you specify either or both the ARGLIST or USES_VAX_ARGLIST arguments, any value you specify with the FUNC_RETURN argument is recorded in both the procedure descriptor and the procedure signature block.

USES_VAX_ARGLIST
An optional argument to indicate whether the called routine uses a VAX argument list. This argument is ignored unless you also specify TIE=TRUE. By default, $CALL assumes the called routine does not use a VAX argument list. Specify USES_VAX_ARGLIST=TRUE to indicate that the called routine uses a VAX argument list.

SIGNATURE_BLOCK
An optional argument that you can use to supply the address of the call signature block. This argument is ignored unless you also specify TIE=TRUE. Note that you cannot specify a SIGNATURE_BLOCK argument in combination with either of the FUNC_RETURN or USES_VAX_ARGLIST arguments. By default, $CALL generates a call signature block for you when you specify TIE=TRUE, and you can in part control the contents of that signature block with the FUNC_RETURN and USES_VAX_ARGLIST arguments. If you wish to define your own call signature block, do not specify either of the FUNC_RETURN or USES_VAX_ ARGLIST arguments and supply the address of your call signature block with the SIGNATURE_BLOCK argument.

NONSTANDARD
A Boolean value (TRUE or FALSE) that specifies whether $CALL should suppress warning and informational messages concerning nonstandard usage. By default, $CALL issues warning and informational messages to indicate you are using $CALL in a way that violates the OpenVMS Calling Standard or in a way that requires special programming considerations. Specify NONSTANDARD=TRUE if you wish to suppress these messages.

Description

$CALL issues a call to another routine and performs the following actions:

1. Searches a list of linkage pairs referenced in previous invocations of the $CALL and $LINKAGE_PAIR macros. If a linkage pair is already in the list, $CALL uses the linkage pair from the previous invocation. Otherwise, $CALL stores the linkage pair of the called routine in the caller's linkage section and adds the linkage pair to the caller's list. If you use $CALL within a routine defined with the $ROUTINE macro, $CALL and $LINKAGE_PAIR reset the list of linkage pairs for each routine.
2. Allocates stack space for arguments if necessary.
3. Generates instructions to load the arguments to the called routine.
4. Sets the value in the argument information register, R25.

5. Generates the following instruction sequence to perform the actual call based on the location of the linkage pair generated from step 1 and the address specified or defaulted with the LS argument. The register specified with the Rls argument is assumed to point to the base of the linkage section as shown in the following example:

```
LDQ R26, code_address_offset(Rls)         ; load code address
LDQ R27, procedure_descriptor_address_offset(Rls) ; load
                                                  ;  procedure
                                                  ;  descriptor
                                                  ;  address
    JSR R26, R26
```

6. Frees argument stack space, if any, and if the called routine does not return a value on the stack.

Examples

Example 1

```
    $CALL SUB1, Rls=R13, LS=MY_LINKAGE_SECTION
    .BASE R13, MY_LINKAGE_SECTION
    $CALL SUB2

    $ROUTINE SUB3, KIND=STACK, SAVED_REGS=<R2,FP>
    $LINKAGE_SECTION
    .QUAD 1
XX:
    $CODE_SECTION
    MOV R27, R2
    $CALL SUB4, R2
    .BASE R2, $LS
    $CALL SUB5
    $CALL SUB6, -
       ARGS=<XX/A, R0/Q>, -
       SCRATCH_REGS=<R22,R23>
    $CALL SUB7, -
       ARGS=<1/A,2/A,3/A>
    $RETURN
    $END_ROUTINE SUB3
```

Example 2

```
$CALL FOO, ARGS=<A,B,C,D,E,F,G,H>, STACK_RETURN_VALUE=TRUE
                        ; Fetch octaword return value from stack
LDQ R4, 0(SP)           ; low quadword
LDQ R5, 8(SP)           ; high quadword
LDA SP, $STACK_ARG_SIZE+16(SP)  ; RESET STACK
```

$CODE_SECTION

Switches control to the current routine's code section psect.

Format

$CODE_SECTION

Description

$CODE_SECTION switches control to the current routine's code section psect.

Example

```
$CODE_SECTION
```

$DATA_SECTION

Switches control to the current routine's data section psect.

Format

$DATA_SECTION

Description

$DATA_SECTION switches control to the current routine's data section psect.

Example

```
$DATA_SECTION
```

$END_EPILOGUE

Marks the end of an epilogue instruction sequence.

Format

$END_EPILOGUE

Description

You can use the $END_EPILOGUE macro to mark the end of an epilogue instruction sequence that you code yourself within a stack or register routine defined with the $ROUTINE macro. At each point where a STACK or REGISTER routine returns to its caller, the routine must perform a sequence of operations to restore any saved registers and to perform stack-frame management. This sequence is called the epilogue and is described in detail in the *OpenVMS Calling Standard*.

You can use the $RETURN macro to generate a standard epilogue instruction sequence for you. Alternatively, you can code your own sequence. If you code your own epilogue sequence, you must mark the beginning and end of the epilogue sequence with the $BEGIN_EPILOGUE and $END_EPILOGUE macros, respectively.

Note that you must not use $END_EPILOGUE for an epilogue instruction sequence generated by $RETURN. $RETURN invokes $BEGIN_EPILOGUE and $END_EPILOGUE for you.

You may omit the invocation of $END_EPILOGUE if your epilogue sequence occurs at the end of the routine. The $END_ROUTINE macro invokes $END_EPILOGUE for you if you invoke the $BEGIN_EPILOGUE macro without a matching invocation of the $END_EPILOGUE macro. You must invoke $END_EPILOGUE with epilogue sequences that occur in the middle of your routine.

Example

```
$ROUTINE MUMBLE, KIND=REGISTER, SAVE_FP=R1
    :
    :
    :
$BEGIN_EPILOGUE
MOV     R1,FP               ; Restore caller's frame
RET     (R26)               ; Return to caller
$END_EPILOGUE
    :
    :
    :
$END_ROUTINE MUMBLE
```

$END_PROLOGUE

Marks the end of a prologue instruction sequence.

Format

$END_PROLOGUE

Description

$END_PROLOGUE marks the end of a routine prologue instruction sequence that you code yourself. The prologue instruction sequence begins with the first instruction of the routine. There can only be one prologue instruction sequence in a routine. The prologue is described in detail in the *OpenVMS Calling Standard*.

You invoke $END_PROLOGUE after the last instruction in the routine prologue code. The last instruction in the routine prologue is the one that updates FP (R29, the frame pointer) and makes the frame become current. You must use this macro when the routine type is stack or register and you specify STANDARD_PROLOGUE=FALSE to the $ROUTINE macro.

Notes

- Do not use this macro when the $ROUTINE macro generates a standard prologue or when the routine type is null or bound. For example, a standard prologue is generated for you when you specify or leave the default of $ROUTINE's STANDARD_PROLOGUE argument to TRUE.

Example

```
MOV     SP, FP
$END_PROLOGUE
```

$END_ROUTINE

Marks the end of a routine.

Format

$END_ROUTINE [name=routine-name]

Parameter

name
The name of the routine that is ended. This argument is optional. If you specify a name, $END_ROUTINE verifies that the name matches that which was specified with $ROUTINE to begin the routine. If the name does not match, $END_ROUTINE issues a diagnostic message. There is no default. If you omit the name, $END_ROUTINE does not verify a matching name.

Description

You must use this macro at the end of a routine that is defined with the $ROUTINE macro to delimit the end the current routine and to perform final macro processing of the routine.

Example

```
$END_ROUTINE NAME=FOOZLE
```

$LINKAGE_PAIR

Locates or defines a linkage pair in the linkage psect.

Format

$LINKAGE_PAIR name=routine-name, local=boolean

Parameters

name

Name of the linkage pair to define. This argument is required.

local

A Boolean value (TRUE or FALSE) that specifies whether the routine is defined within the module for not. The default is to store a linkage pair for a global routine. You must specify LOCAL=TRUE to store a linkage pair for a routine that is not globally visible.

Description

You can invoke this macro to locate or define a linkage pair in the linkage psect. The linkage pair is added to a list for later use and retrieval by the $CALL and $LINKAGE_PAIR macros. Thus, only the first invocation of $LINKAGE_PAIR (or $CALL) for a given routine to be called results in a linkage pair being stored. $LINKAGE_PAIR and $CALL use the same list of linkage pairs. $LINKAGE_PAIR restores the current psect when it is done.

$LINKAGE_PAIR defines the symbol $LP. This symbol value is the address within the linkage section of the linkage pair for the specified routine to call.

Notes

- Because the $CALL macro invokes the $LINKAGE_PAIR macro for you, you do not need to use $LINKAGE_PAIR when you are using $CALL. You may wish to use $LINKAGE_PAIR when you are not using $CALL or when you require a specific ordering or placement of linkage pairs within the linkage section.

Example

```
$LINKAGE_PAIR SUB1          ; define linkage pair in linkage section
LDQ R26, $LP
LDQ R27, $LP+8
JSR R26, R26
```

$LINKAGE_SECTION

$LINKAGE_SECTION switches control to the current routine's linkage section psect.

Format

$LINKAGE_SECTION

Description

$LINKAGE_SECTION switches control to the current routine's linkage section psect.

Example

```
$LINKAGE_SECTION
```

$OPDEF

Used to define opcodes.

Format

$OPDEF MNEMONIC, FORMAT, ENCODING [,DEFAULTS]

Parameters

MNEMONIC

MNEMONIC is the mnemonic name by which the instruction is called. You may optionally specify a qualifier list separated from the name by a slash (/). A qualifier list is a sequence of one or more letters or digits with no intervening spaces or separators.

FORMAT

FORMAT is one of the following arguments:

Format	Description
MEMORY	Standard memory format instructions.
MEMORY_FUNCTION	Memory format instructions with a function code.
JUMP	Memory format instructions formatted like jump instructions.
BRANCH	Standard branch format instructions.
OPERATE	Standard operate instructions.
FLOATING_OPERATE	Standard floating-point operate instructions.
PAL	Standard PALcode instructions.
<CUSTOM=operand_type_list>	Custom format.

With the CUSTOM format, you may optionally specify a list of the types of operands the instruction is to accept. If you specify a list of operand types, you must enclose the entire FORMAT argument within angle brackets, and you must specify the operand types in the order they are to be used with the instruction. $OPDEF supports the following operand types:

IREGISTER	Integer register, such as R5 or SP.
FREGISTER	Floating-point register, such as F7.
LITERAL	Integer literal, such as #123 or 32767.
LIT_IREG	Integer literal, such as #123 or 32767, or integer register, such as R5 or SP.
INDIRECT	Indirect integer register notation such as (R7).
DISPLACEMENT	Indirect integer register notation with an integer literal displacement, such as FOO(R12).
BRANCH_OFFSET	Label or address expression, such as L1.

For example:

```
FORMAT=<CUSTOM=IREGISTER,DISPLACEMENT>
```

The following example shows the definition of the ADDQ instruction, which takes either an integer register or literal as its second argument:

```
$OPDEF ADDQ,                                        -
    FORMAT=<CUSTOM=IREGISTER,LIT_IREG,IREGISTER>, -
    ENCODING=<26:31=^x10,                           -
        21:25=%OP1,                                 -
        12:20=%OP2,                                 -
        5:11=^x20,                                  -
        0:4=%OP3>
```

For a detailed description of instruction formats, see the *Alpha Architecture Reference Manual*.

ENCODING

ENCODING is the numeric encoding for the instruction opcode. The default radix is decimal, as is the case for all assembler constants. Prefix the value with ^X for hexadecimal. Certain instruction formats allow multipart encoding:

Format	Encoding Description
MEMORY_FUNCTION	Specify the base opcode, followed by a dot, followed by the function code. For example: `ENCODING=^X10.F000`
JUMP	Specify the base opcode, optionally followed by a dot, and the hardware-hint bits optionally followed by a dot and the software-hint bits. For example: `ENCODING=^X1A.2.1`
OPERATE	Specify the base opcode, followed by a dot, followed by the function code. For example: `ENCODING=^X12.3C`
FLOATING_OPERATE	Specify the base opcode, followed by a dot and the function code. For example: `ENCODING=^X17.02B`

PAL	Specify the base opcode, optionally followed by a dot and the function code. Omit the function code for a generic PAL instruction that accepts an integer-expression argument. For example:
	`ENCODING=^X0.0080`
	`ENCODING=^X19`

CUSTOM	Specify a comma-separated list of bit ranges and values to place in those bit ranges. For example:
	`ENCODING = < 26:31=^X14, 21:25=%OP1, -`
	`16:20=%OP2.REG, 0:15=%OP2.DISP >`

For CUSTOM format instructions, specify the ENCODING argument as a comma-separated list of bit ranges and values to place in those bit ranges. Enclose the list within angle brackets.

Specify a bit range as start:end where start and end are integer constant expressions. For a 1-bit bit range, start and end are equal. Bit positions range from 0 to 31. Place an equal sign (=) after the bit-range specifier followed by the value you wish to put in the bit range. You can place either a constant integer expression or an operand into the bit range. Start and end expressions and integer constant expressions must not reference any external symbols or symbols not yet defined in the assembly. $OPDEF evaluates these expressions at the time that it defines the instruction as opposed to when the defined instruction is referenced.

Operand names are of the form %OPn, where n is the ordinal number of the operand as specified from left to right with the FORMAT argument.

For the IREGISTER, FREGISTER, and INDIRECT operands, $OPDEF places the 5-bit register number into the bit positions you specify.

For LITERAL operands, $OPDEF places the literal value of the operand into the bit positions you specify. Operand literal values can be up to 32 bits long. The most significant bits are truncated if the size of the operand literal value exceeds the bit range you specify. Forward and external references are allowed.

For LIT_IREG operands, $OPDEF places either a literal value or a 5-bit register number into the bit positions you specify. If a literal, the low bit is 1, and the literal value is placed in the upper bits. If an integer register, the low four bits are 0, and the high five bits contain the register number.

For DISPLACEMENT operands, $OPDEF defines two parts: a 5-bit register number and a displacement value that can be up to 32 bits long. The most significant bits are truncated from the displacement value if the size of the displacement value exceeds the bit range you specify. You can reference the register number by placing .REG after the operand name. For example: %OP2.REG. Similarly, you can reference the displacement value by placing .DISP after the operand name. For example: %OP2.DISP. Forward references are allowed. Relocatable expressions are not allowed.

For BRANCH_OFFSET operands, $OPDEF stores the signed longword offset between the next instruction and the specified address in the bit positions you specify. The address expression specified for a BRANCH_OFFSET operand can be a backward or forward reference to a location within the current psect. It cannot be an external address or an address in a different psect. The resulting offset can be up to 32 bits in size. If the size of the offset exceeds the bit range you specify, the most significant bits are truncated.

$OPDEF fills any bit positions you leave unspecified with zeros.

DEFAULTS

DEFAULTS is an optional list of operand defaults of the form <%OPn=value, ...>, where n is the number of the operand to which the value is to apply as a default. Operand defaults may be specified in any order. If you specify a default for one or more operands, you need not specify a default for all operands.

The following example specifies a default of R31 for the first instruction argument:

```
$OPDEF RET, FORMAT=<CUSTOM=IREGISTER,INDIRECT>,   -
    ENCODING=<26:31=^x1A,                         -
        21:25=%OP1,                               -
        16:20:%OP2,                               -
        14;14=^x2,0:13=0>,                        -
    DEFAULTS=<%OP1=R31>
```

Description

You can use the $OPDEF macro to define your own opcodes.

$OPDEF defines a macro using an unqualified version of the mnemonic name you specify. When this macro is invoked with the instruction qualifiers you specify when you define it with $OPDEF (if any), it expands to the instruction representation you have defined with $OPDEF. You can specify the qualifiers in any order as long as the combination of qualifiers you use is the same.

Other uses of the mnemonic name remain valid provided you do not use the mnemonic name as a macro name in a .MACRO directive. For instance, you can use the same mnemonic name in the opcode field with different or no qualifiers. If the qualifiers (or absence thereof) do not match those specified in your $OPDEF instruction definition, the macro defined by $OPDEF processes as though you had not defined an instruction by that mnemonic name. To do so, it expands to a single statement. This expansion statement is identical to the mnemonic-name macro invocation statement, except it is processed in a context that prevents the mnemonic-name macro from expanding recursively. Instead, the statement is processed as a normal, MACRO–64 instruction statement. In this case, you may notice references to the mnemonic-name macro expansion in a MACAUXMSG diagnostic message if the instruction statement contains errors.

For instance, if you define a STQ/P instruction using $OPDEF, you can still use the STQ instruction without the /P qualifier. If you do, and your STQ instruction statement contains an error, the assembler generates a MACAUXMSG message indicating that the error occurred during the expansion of macro STQ. Aside from the fact that the STQ instruction is processed in the context of the expansion of the STQ macro, $OPDEF's definition of the STQ/P instruction has no effect on your use of the STQ instruction.

Example

```
$OPDEF MNEMONIC=BANG, FORMAT=PAL, -
       ENCODING=^X0.0099
```

.PACKED

Packed decimal string storage macro.

Format

.PACKED decimal-string[,symbol]

Parameters

decimal-string
A decimal number from 0 to 31 digits long with an optional sign. Digits can be in the range 0 to 9.

symbol
An optional symbol that is assigned a value equivalent to the number of decimal digits in the string. The sign is not counted as a digit.

Description

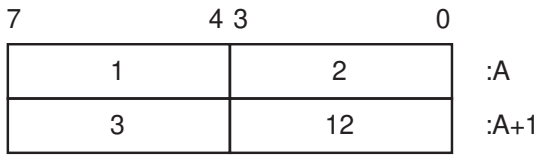
.PACKED generates packed decimal data with two digits per byte. Packed decimal data is useful in calculations requiring exact accuracy. It is operated on by the decimal string instructions.

A packed decimal string is a contiguous sequence of bytes in memory. It is specified by two attributes: the address A of the first byte and a length L, which is the number of digits in the string and not the length of the string in bytes. The bytes of a packed decimal string are divided into two, 4-bit fields (nibbles). Each nibble except the low nibble (bits 3:0) of the last (highest-addressed) byte must contain a decimal digit. The low nibble of the highest-addressed byte must contain a sign. The representation for the digits and sign is indicated as follows:

Digit or Sign	Decimal	Hexadecimal
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10,12,14, or 15	A,C,E, or F
–	11 or 13	B or D

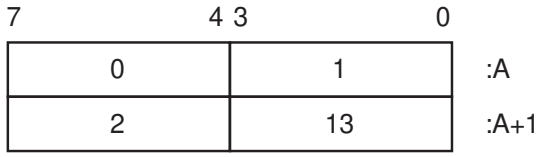
The preferred sign representation is 12 for plus (+) and 13 for minus (–). The length L is the number of digits in the packed decimal string (not counting the sign); L must be in the range 0 to 31. When the number of digits is odd, the digits and the sign fit into a string of bytes whose length is defined by the following equation: `L/2(integer part only) + 1`. When the number of digits is even, it is required that an extra 0 appear in the high nibble (bits 7:4) of the first byte of the string. Again, the length in bytes of the string is L/2 + 1.

The address A of the string specifies the byte of the string containing the most-significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. Thus, +123 has a length of 3 and is represented as follows:



ZK–6463A–GE

The packed decimal number −12 has a length of 2 and is represented as follows:



ZK–6464A–GE

Example

```
.PACKED -12,PACKED_SIZED        ; PACKED_SIZE gets value of 2
.PACKED +500
.PACKED 0
.PACKED -0,SUM_SIZE             ; SUM_SIZE gets value of 1
```

$PROCEDURE_DESCRIPTOR

Defines a procedure descriptor structure at the current psect and offset.

Format

$PROCEDURE_DESCRIPTOR

Description

The arguments for the $PROCEDURE_DESCRIPTOR macro are the same as the $ROUTINE macro, with the following exceptions:

- The ENTRY argument is required.
- There are no CODE_SECTION, LINKAGE_SECTION, DATA_SECTION, DATA_SECTION_POINTER, or STANDARD_PROLOGUE arguments.
- There is an additional END_PROLOGUE argument. This argument must be a label that you define at the end of the routine's prologue sequence. This argument is required for stack and register procedure types.

Notes

- Because the $ROUTINE macro invokes the $PROCEDURE_DESCRIPTOR macro for you, you do not need to use the $PROCEDURE_DESCRIPTOR macro if you use $ROUTINE. You may wish to use $PROCEDURE_DESCRIPTOR when you are not using $ROUTINE.

Example

```
$PROCEDURE_DESCRIPTOR p1,           -
                      KIND=NULL,    -
                      ENTRY=p1_entry
```

$RESET_LP_LIST

Clears the list of linkage pairs maintained by the $LINKAGE_PAIR and $CALL macros.

Format

$RESET_LP_LIST

Description

The $LINKAGE_PAIR and $CALL macros maintain an assembly-time list of linkage pairs that the $LINKAGE_PAIR macro has stored in the linkage section. This list enables $LINKAGE_PAIR and $CALL to use the same linkage pair for multiple routine calls to the same routine. You can clear this list of linkage pairs with the $RESET_LP_LIST macro.

Under normal circumstances, you do not need to clear the linkage-pair list. Some of the times when you do are as follows:

- When you change the psect assigned to the linkage section.
- If distance from the linkage section of your current routine falls more than 32K bytes beyond where the linkage pair has been stored.

Example

```
; Define the linkage section psect for routines A & B
$LINK$ = "AB_LINK,NOEXE,OCTA"

$ROUTINE A, KIND=STACK
.BASE  R27, $LS
$CALL D ; Linkage pair is stored in A's linkage
  ;  section and put on the assembly-time list
$RETURN
$END_ROUTINE A

$ROUTINE B, KIND=STACK
.BASE  R27, $LS
$CALL D ; Linkage pair is found on the list,
  ;  and used from A's linkage section
$RETURN
$END_ROUTINE B

; Define a different linkage section for routine C
$LINK$ = "C_LINK,NOEXE,OCTA"

; Linkage pairs that are on the list are in A & B's linkage
; section, which is not easily accessible by C.  Therefore,
; clear the list.
$RESET_LP_LIST

$ROUTINE C, KIND=STACK
.BASE  R27, $LS
$CALL D ; Linkage pair is stored in C's linkage
  ;  section and put on the assembly-time list
$RETURN
$END_ROUTINE B
```

$RETURN

Generates a standard epilogue instruction sequence.

Format

$RETURN

Description

Generates a standard epilogue instruction sequence when used within a stack or register routine defined with the $ROUTINE macro. The epilogue sequence generated by $RETURN restores any registers you specify with the SAVED_REGS argument to $ROUTINE and performs stack frame management as necessary. You can use $RETURN whether or not you specify STANDARD_PROLOGUE as TRUE or accept the default.

You can use $RETURN any number of times within a given stack or register routine to affect a return to the calling routine.

You must not use the $BEGIN_EPILOGUE or $END_EPILOGUE macros for an epilogue sequence generated by $RETURN. $RETURN invokes $BEGIN_EPILOGUE and $END_EPILOGUE for you.

Example

```
$ROUTINE FOOZLE, KIND=REGISTER, SAVE_FP=R1
    :
    :
    :
$RETURN
$END_ROUTINE FOOZLE
```

$ROUTINE

Defines the current routine and creates a context for the routine.

Format

```
$ROUTINE  NAME=routine name -
          ALIASES=alias names -
          LOCAL=boolean value -
          STANDARD_PROLOGUE=boolean value -
          ENTRY=code entry point -
          CODE_SECTION=code section psect name -
          DATA_SECTION=data section psect name -
          DATA_SECTION_POINTER=boolean value -
          LINKAGE_SECTION=linkage section psect name -
          KIND=routine type-
          HANDLER_REINVOKABLE=boolean value -
          BASE_REG_IS_FP=boolean value-
          REI_RETURN=boolean value -
          STACK_RETURN_VALUE=boolean value -
          RSA_OFFSET=integer value -
          SAVE_FP=register name -
          SAVE_RA=return address register name -
          SIZE=numeric value -
          SAVED_REGS=list of registers -
          HANDLER=exception handler address -
          HANDLER_DATA=data address for exception handler -
          SYNCH_EXCEPTIONS=boolean value-
          PROC_VALUE=procedure value -
          ENVIRONMENT=environment value -
          FUNC_RETURN=function return type -
          ARGLIST=argument type list -
          USES_VAX_ARGLIST=boolean value -
          OVERRIDE_FLAGS=procedure descriptor flags -
          DEFAULT_SIGNATURE=boolean value -
          COMMON_BASE=list of registers -
          TARGET_INVO=boolean value -
          EXCEPTION_MODE=mode -
```

Parameters

NAME

The name of the routine. This argument is required for all procedure kinds. There is no default. For example:

```
NAME=FOOZLE
```

ALIASES

List of alias names for the routine. This argument is optional for all procedure types. There is no default. For example:

```
ALIASES=<FOO,BAR,FOOBAR,foo,bar,foobar,Foo,Bar,FooBar>
```

LOCAL

Boolean value indicating whether the routine is local (TRUE) or externally visible (FALSE). This argument is optional for all procedure kinds. The default is FALSE. For example:

```
LOCAL=TRUE
```

STANDARD_PROLOGUE

Specifies a Boolean value to indicate whether $ROUTINE should generate a standard instruction prologue sequence at the beginning of the routine's code section. This argument is optional and valid only with REGISTER and STACK procedures. If the procedure type is stack or register, the default is TRUE and $ROUTINE generates a standard prologue sequence. The prologue sequence generated by $ROUTINE saves the registers you specify with the SAVED_REGS argument and performs stack-frame management as necessary.

If you also specify BASE_REG_IS_FP=FALSE, the standard prologue sequence generated by $ROUTINE makes a copy of the procedure descriptor address that is in R27 upon entry into R29 (FP). While you cannot change the value in R29 before the epilogue, you can use R29 as a working, linkage-section register. If you specify the STANDARD_PROLOGUE argument as FALSE, you must code your own prologue sequence and mark the end of the prologue with the $END_PROLOGUE macro. Whether or not you specify STANDARD_PROLOGUE as TRUE or accept the default, you can generate a standard epilogue sequence for stack and register procedures with the $RETURN macro. For example:

```
STANDARD_PROLOGUE=FALSE
```

ENTRY

The name of the code-entry point. This argument is the code entry-point label that $ROUTINE defines for you at the beginning of the code section for the routine. If this argument is omitted, $ROUTINE generates a label. For example:

```
ENTRY=FOOZLE_ENTRY
```

CODE_SECTION

The psect name and attributes of the code section. This argument is optional for all procedure kinds. If omitted, the default is the name and attributes defined by the $CODE$ lexical string symbol. If you specify a name and attributes for the CODE_SECTION argument, $ROUTINE redefines the $CODE$ lexical string symbol such that the specified values become the new default. Initially, $CODE$ is defined as follows:

```
$CODE$ = "$CODE$,OCTA,PIC,CON,REL,LCL,SHR,EXE,NORD,NOWRT"
```

Since you must delimit the psect name and attributes using commas, be sure to enclose this argument within angle brackets to avoid having the assembler interpret the name and attributes as different arguments to $ROUTINE. For example:

```
CODE_SECTION=<MY_CODE,EXE,QUAD,NOWRT>
```

DATA_SECTION

The psect name and attributes of the data section. This argument is optional for all procedure kinds. If omitted, the default is the name and attributes defined by the $DATA$ lexical string symbol. If you specify a name and attributes for the DATA_SECTION argument, $ROUTINE redefines the $DATA$ lexical string symbol such that the specified values become the new default. Initially, $DATA$ is defined as follows:

```
$DATA$ = "$DATA$,OCTA,NOPIC,CON,REL,LCL,NOSHR,NOEXE,RD,WRT"
```

Since you must delimit the psect name and attributes using commas, be sure to enclose this argument within angle brackets to avoid having the assembler interpret the name and attributes as different arguments to $ROUTINE. For example:

```
DATA_SECTION=<MY_DATA,NOEXE,QUAD,RD,WRT>
```

DATA_SECTION_POINTER

Boolean value indicating whether $ROUTINE should store a pointer to the data section in the linkage section and define $DP as the address of that pointer. This argument is optional for all procedure kinds. The default is FALSE. For example:

```
DATA_SECTION_POINTER=TRUE
```

You can use the DATA_SECTION_POINTER argument as follows:

```
        $ROUTINE TALLY_HO, DATA_SECTION_POINTER=TRUE
        $DATA_SECTION
TALLY:  .QUAD 0
        $CODE_SECTION
        .BASe       R27, $LS  ; Inform assembler that R27->$LS
        LDQ         R1, $DP   ; R1->$DS
        .BASE R1,$DS          ;Inform assembler that R1-$DS
        LDQ         R0, TALLY ; R0<-TALLY
        LDA         R0, 1(R0) ; R0<-R0++
        STQ         R0, TALLY ; TALLY<-R0
        RET         (R26)     ; Return
        $END_ROUTINE TALLY_HO
```

In this example, the DATA_SECTION_POINTER argument is specified in order to obtain linkage to the data section. The first LDQ instruction loads R1 with the pointer to the data section that $ROUTINE stores in the linkage section. The next three instructions increment the value in the TALLY variable in the data section. Finally, the routine returns the incremented value to its caller in R0.

LINKAGE_SECTION

The psect name and attributes of the linkage section. This argument is optional for all procedure kinds. If omitted, the default is the name and attributes defined by the $LINK$ lexical string symbol. If you specify a name and attributes for the LINKAGE_SECTION argument, $ROUTINE redefines the $LINK$ lexical string symbol such that the specified values become the new default. Initially, $LINK$ is defined as follows:

```
$LINK$ = "$LINK$,OCTA,NOPIC,CON,REL,LCL,NOSHR,NOEXE,RD,NOWRT"
```

Since you must delimit the psect name and attributes using commas, be sure to enclose this argument within angle brackets to avoid having the assembler interpret the name and attributes as different arguments to $ROUTINE. For example:

```
LINKAGE_SECTION=<MY_LINK,NOEXE,QUAD,RD,NOWRT>
```

KIND

Specifies the kind of routine being defined. This must be one of the following: STACK, REGISTER, NULL, or BOUND. This is an optional argument. The default is NULL. For example:

```
KIND=STACK
```

HANDLER_REINVOKABLE

Specifies a Boolean value to indicate whether the condition handler can be re-invoked. This argument is optional and valid only with STACK and REGISTER procedures. It defaults to FALSE if no value is specified and the procedure kind is STACK or REGISTER. Note that this argument is only valid if a value is also specified for the HANDLER argument. For example:

```
HANDLER_REINVOKABLE=TRUE
```

BASE_REG_IS_FP

Specifies a Boolean value to indicate whether register R29 (FP) is used as the frame-pointer base register (TRUE) or not (FALSE). If specified as FALSE, R29 must be used to hold the address of the procedure descriptor (or the address of the address of the procedure descriptor—see the *OpenVMS Calling Standard*. You can use R29 to hold a working copy of the linkage-section address passed in R27. In addition, your prologue and epilogue instruction sequences can be shorter and more efficient. However, you cannot make standard calls to other routines if BASE_REG_IS_FP is specified as FALSE. This argument is optional and valid only with stack and register procedure kinds. It defaults to TRUE if the procedure type is stack or register. For example:

```
BASE_REG_IS_FP=FALSE
```

REI_RETURN

Specifies a Boolean value to indicate whether this routine returns using an REI instruction. This argument is optional and valid only with STACK, REGISTER, and NULL procedure kinds. It defaults to FALSE if the procedure kind is STACK, REGISTER, or NULL. For example:

```
REI_RETURN=TRUE
```

STACK_RETURN_VALUE

This argument is obsolete. Do not specify this argument.

RSA_OFFSET

An integer value specifying the stack offset (in bytes) of the register save area. This argument is optional and valid only for STACK procedures. If you specify BASE_REG_IS_FP as TRUE, the value you specify with RSA_OFFSET must be at least 8. RSA_OFFSET defaults to 8 if BASE_REG_IS_FP is specified as TRUE, 0 otherwise. For example:

```
RSA_OFFSET=32
```

SAVE_FP

The register that contains a copy of the value of FP (R29) upon entry to this routine. The prologue instruction sequence must copy FP to the register specified by SAVE_FP and the epilogue instruction sequence(s) must restore FP from the register specified by SAVE_FP. This argument is required and only valid for REGISTER procedures. There is no default. For example:

```
SAVE_FP=R1
```

SAVE_RA

The register that contains the return address. This argument is optional and only valid for REGISTER procedures. If SAVE_RA is not R26, the prologue instruction sequence must copy R26 to the register specified by SAVE_RA and the epilogue instruction sequence(s) must use the return address stored in the

register specified by SAVE_FP to affect its return to caller. SAVE_RA defaults to R26 if the procedure kind is REGISTER. For example:

```
SAVE_RA=R22
```

SIZE

A numeric value that must be a multiple of 16 specifying the size of the fixed-stack area in bytes. This parameter is valid only for REGISTER and STACK procedure kinds. It defaults to the minimum value possible given the other arguments you specify or default. $ROUTINE computes the amount of stack storage needed for the register save area (if any) and defines the $RSA_END symbol to be the offset of the first byte beyond the register save area. If you wish to allocate a fixed area of stack beyond the register save area, you can specify an expression with the SIZE argument that includes the term $RSA_END plus the amount of fixed-stack storage you need for your application. For example:

```
SIZE=$RSA_END+32
```

SAVED_REGS

A list of registers saved on the stack by the prologue code of this routine. It is valid only for STACK procedures and you must specify at least FP (R29) with this argument. It defaults to FP (R29) for STACK procedures. For example:

```
SAVED_REGS=<R5,FP,F2,F3,F4>
```

The OpenVMS Calling Standard specifies that registers R0, R1, R28, R30 (SP), and R31 must never be saved and restored. If you specify these registers with the SAVED_REGS argument, the $ROUTINE macro issues a diagnostic warning message.

HANDLER

The address of an exception handler. It is optional and valid only for STACK and REGISTER procedure kinds. By default, the procedure is defined not to have an exception handler. For example:

```
HANDLER=MY_HANDLER
```

HANDLER_DATA

The address of data for the specified handler, if any. This argument is optional and valid only for stack and register procedure kinds and has no default value. You cannot specify a HANDLER_DATA argument if you do not specify the HANDLER argument. For example:

```
HANDLER_DATA=MY_HANDLER_DATA
```

SYNCH_EXCEPTIONS

An argument to indicate whether exceptions must be synchronized or not. This argument is optional with STACK and REGISTER routines and is not allowed with other kinds of routines. This argument defaults to TRUE if you specify an exception handler with the HANDLER argument. Otherwise, it defaults to FALSE. When this argument is TRUE and you specify or accept the default STANDARD_PROLOGUE=TRUE, $ROUTINE generates a TRAPB instruction as part of the standard prologue sequence. In addition, when this argument is true, the $RETURN macro generates a TRAPB instruction as part of the standard epilogue sequence. When this argument is FALSE, neither $ROUTINE nor $RETURN generate TRAPB instructions.

PROC_VALUE

The procedure value of a bound procedure's parent. This argument is required for BOUND procedures and is invalid for all other procedure kinds. For example:

```
PROC_VALUE=PARENT_PROC
```

ENVIRONMENT

Specifies an environment value. This parameter is optional and valid only for BOUND procedures. It has no default value. For example:

```
ENVIRONMENT=0
```

FUNC_RETURN

Specifies the function return type. This argument is optional and valid for all procedure kinds. If specified, it must be one of the following: I64, D64, I32, U32, FF, FD, FG, FS, FT, FFC, FDC, FGC, FSC, or FTC. These values correspond to those listed in Table 3-7 of the *OpenVMS Calling Standard* that have an additional "RASE$K_FR_" prefix. There is no default. For example:

```
FUNC_RETURN=U32
```

ARGLIST

Argument type list. This argument is optional and valid for all procedure kinds. If the argument list contains one or more elements, each of the first six elements must be one of the following: Q, I32, U32, FF, FD, FG, FS, or FT. The seventh and subsequent arguments (if any) must be either I32 or Q.

These values correspond to the PSIG$K_RA_* and MASE$K_MA_* signature encodings in Table 3-6 of the *OpenVMS Calling Standard*. There is no default. If you specify this argument, $ROUTINE generates a procedure signature block. For example:

```
ARGLIST=<Q,I32,FF,FF,U32>.
```

USES_VAX_ARGLIST

Specifies a Boolean value indicating whether the routine uses a VAX argument list. This argument is optional for all procedure kinds and defaults to FALSE. If you specify this argument, $ROUTINE generates a procedure signature block. For example:

```
USES_VAX_ARGLIST=TRUE
```

OVERRIDE_FLAGS

Specifies overriding flags for the PDSC$W_FLAGS field in the procedure descriptor. This argument is optional and valid for all procedure kinds. However, it is required for BOUND procedures when the parameter specified with the PROC_VALUE argument is an external or forward reference. There is no default. For example:

```
OVERRIDE_FLAGS=PARENT_FLAGS
```

DEFAULT_SIGNATURE

Specifies a Boolean value to indicate whether the standard procedure signature is used. TRUE means to use the standard signature. It is optional and valid for all procedure kinds. The default is FALSE if you specify either the ARGLIST or USES_VAX_ARGLIST arguments. Otherwise, the default is TRUE. Note that this argument must be FALSE or blank if you specify either the ARGLIST or USES_VAX_ARGLIST arguments. For example:

```
DEFAULT_SIGNATURE=TRUE
```

COMMON_BASE

An argument to specify one or more base registers that are used in common with other routines. This argument is optional for all routine kinds. By default, $ROUTINE invalidates any previous .BASE directives that may be in effect when you invoke $ROUTINE. In this way, you are prevented from inadvertently processing the second or subsequent routines in a module with .BASE directives in effect that apply only to a previous routine. However, you may wish to share a common base register assignment between a number of routines. To do so, you can reissue the appropriate .BASE directive or directives after each invocation of $ROUTINE. Alternatively, you can specify one or more common base registers with the COMMON_BASE argument, and enter the appropriate .BASE directive or directives only once at the beginning of the module. Specify the value for the COMMON_BASE argument as a list of integer registers. For example:

```
COMMON_BASE=<R5,R13>
```

In this example, $ROUTINE invalidates any previous .BASE directives except those for registers R5 and R13. Previous .BASE directives for registers R5 and R13 are unaffected.

TARGET_INVO

Specifies a Boolean value indicating whether or not the exception handler for this procedure is invoked when this procedure is the target of an invocation unwind. (TARGET_INVO=TRUE) causes the exception handler to be invoked during an unwind. The default is TARGET_INVO=FALSE.

EXCEPTION_MODE

An argument to specify one of the following exception modes with STACK and REGISTER procedures:

- SIGNAL—raise all exceptions except underflow.
- SIGNAL_ALL—raise all exceptions.
- SILENT—raise no exceptions.
- FULL_IEEE—only raise exceptions as per IEEE control bits.

The default is EXCEPTION_MODE=SIGNAL.

Description

$ROUTINE defines a routine, makes it the current routine, and performs the following actions:

- Creates and associates a linkage section, code section, and data section with the routine.
- Defines a procedure descriptor and optional signature block in accordance with the values of macro arguments.
- Optionally stores a pointer to the data section within the linkage section.
- Creates the following numeric and lexical symbols:

Symbol	Description
$CS	Address at start of the current routine's code section.
$DS	Address at start of the current routine's data section.

$DP	Optional address of a pointer to the current routine's data section. This symbol has a value that is an address in the current routine's linkage section at which the $ROUTINE macro has placed the address of the data section ($DS) as follows:
	`$DP = .`
	`.ADDRESS $DS`
	$DP enables you to access the data area of the current routine from its linkage section.

$LS	Address of the current routine's linkage section.
$SIZE	Size of fixed area of stack frame of the current routine. This symbol is valid only with STACK and REGISTER routines.
$RSA_OFFSET	The offset within the fixed-stack area to the register save area. This symbol is valid only with STACK routines.
$RSA_END	The offset within the fixed-stack area to the the first byte beyond the end of the register save area (if any).
$CODE$	A lexical string symbol that defines the routine's code psect name and attributes.
$DATA$	A lexical symbol that defines the routine's data psect name and attributes.
$LINK$	A lexical string symbol that defines the routine's linkage psect name and attributes.

- Optionally generates a standard instruction prologue sequence at the beginning of the code section.

  If you specify /NAMES=AS_IS on the command line, all but the last three of these symbols are defined in both complete uppercase and complete lowercase. These symbols are intended for your use outside of the macros themselves. For example, the values of these numeric symbols may be useful as a mnemonic when coding an instruction with a register as in the following example:

  ```
  lda   SP,-$SIZE(SP)
  ```

  The last three symbols, $CODE$, $DATA$, and $LINK$, are only defined in uppercase. They are used by the $ROUTINE macro for the default code, data, and linkage section psect names and attributes. You can define these symbols before invoking $ROUTINE to alter the default program sections as follows:

  - $CODE$ = "MY_CODE,EXE,OCTA,SHR,NORD,NOWRT,GBL"
  - $DATA$ = "MY_DATA,NOEXE,OCTA,NOSHR,RD,WRT,GBL"
  - $LINK$ = "MY_LINK,NOEXE,OCTA,SHR,RD,NOWRT,GBL"

These statements cause $ROUTINE to use the previous psect names and attributes by default. If you specify any of the CODE_SECTION, DATA_SECTION, or LINKAGE_SECTION arguments in your invocation of $ROUTINE, $ROUTINE uses the psect name and attributes specified with the argument.

In addition, $ROUTINE redefines the corresponding $CODE$, $DATA$, or $LINK$ lexical string symbol to the value you specify when you specify any of the CODE_SECTION, DATA_SECTION, or LINKAGE_SECTION arguments with $ROUTINE.

Example

```
$ROUTINE MAIN1, KIND=NULL

$ROUTINE MAIN1,         -
    KIND=STACK,         -
    SIZE=48,            -
    SAVED_REGS=<R2,FP,F5>
```

A

MACRO–64 Alpha AXP Architecture Quick Reference

This appendix provides figures and tables showing the data types and addressing capabilities of the Alpha AXP architecture. The information is derived from the *Alpha AXP Achitecture Quick Reference Guide*. Minor changes have been made to reflect the usage of the Alpha AXP architecture that is specific to MACRO–64.

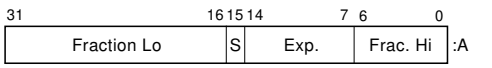
For more information, see the *Alpha Architecture Reference Manual* and the *OpenVMS Calling Standard*.

Figure A–1 shows integer and floating-point data structures. Figure A–2 shows instruction formats.

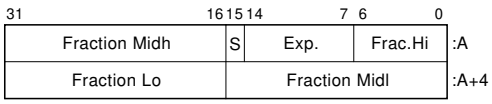
Figure A–1 Data Structures

Data Structures

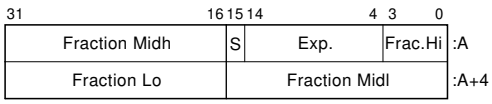
F_floating Datum



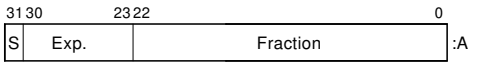
D_floating Datum



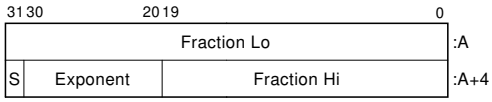
G_floating Datum



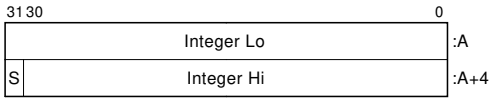
S_floating Datum



T_floating Datum



Quadword Integer Datum



Longword Integer Datum

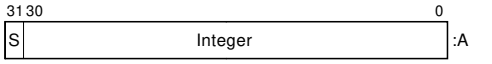
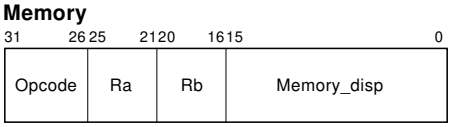


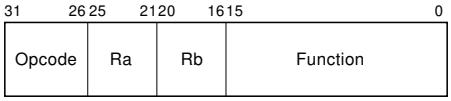
Figure A–2 Instruction Formats

Instruction Formats

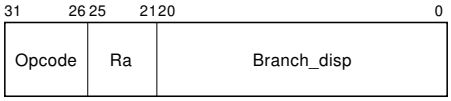
Memory



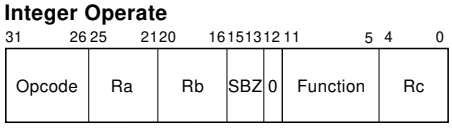
Memory with Function Code

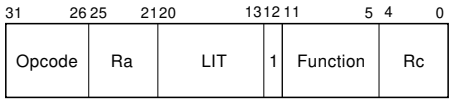


Branch

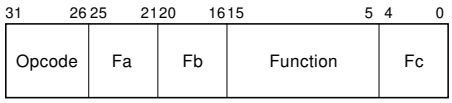


Integer Operate

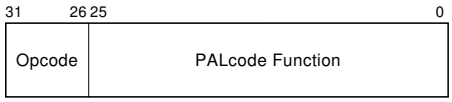




Floating-Point Operate



PALcode



A.1 Register Usage Conventions

Table A–1 lists the register usage conventions for OpenVMS AXP. MACRO–64 recognizes FP and SP as register synonyms, but does not recognize AI, RA, or PV as register synonyms.

Table A–1 Register Usage Conventions for OpenVMS AXP

R0		Int func ret value
R1		Scratch
R2–R15		Saved
R16–R21		Argument
R22–R24		Scratch
R25	AI	Argument information
R26	RA	Return address
R27	PV	Procedure value
R28		Volatile scratch
R29	FP	Stack frame base
R30	SP	Stack pointer
R31		Zero
F0		F-P function ret value
F1		F-P complex func ret value
F2–F9		Saved
F10–F15		Scratch
F16–F22		Argument
F23–F30		Scratch
F31		Zero

A.2 Instruction Operand Notation

Table A–2 shows the notation for instruction operands. The notation format is as follows:

<name>.<access type><data type>

Table A–2 Instruction Operand Notation

<name>:	
disp	Displacement field
fnc	PALcode function field
Ra	Integer register operand in the Ra field
Rb	Integer register operand in the Rb field
#b	Integer literal operand in the Rb field
Rc	Integer register operand in the Rc field
Fa	Floating-point register operand in the Ra field
Fb	Floating-point register operand in the Rb field
Fc	Floating-point register operand in the Rc field
<access type>:	
a	The operand is used in an address calculation to form an effective address. The data-type code that follows indicates the units of addressability (or scale factor) applied to this operand when the instruction is decoded.
i	The operand is an immediate literal.
m	The operand is both read and written.
r	The operand is read only.
w	The operand is write only.

(continued on next page)

Table A–2 (Cont.) Instruction Operand Notation

<data type>:	
b	Byte
f	F_floating
g	G_floating
l	Longword
q	Quadword
s	IEEE single floating (S_floating)
t	IEEE double floating (T_floating)
w	Word
x	The data type is specified by the instruction.

A.3 Instruction Qualifier Notation

Table A–3 shows the notation for instruction qualifiers.

Table A–3 Instruction Qualifier Notation

/Qualifier	Meaning
C	Chopped rounding
D	Dynamic rounding (mode determined by FPCR<DYN>)
I	Inexact result enable
M	Minus infinity rounding
S	Software completion enable
U	Floating underflow enable
V	Integer overflow enable

A.4 F-P Control Register (FPCR) Format

Table A–4 shows the format for the F-P control register.

Table A–4 F-P Control Register (FPCR) Format

Bits	Symbol	Meaning
63	SUM	Bitwise OR of <57:52>
62:60	RAZ/IGN	Read as zero; ignored when written
59:58	DYN	IEEE rounding mode selected:
		00 Chopped
		01 Minus infinity
		10 Normal rounding
		11 Plus infinity
57	IOV	Integer overflow of destination precision
56	INE	Floating mathematically inexact result
55	UNF	Floating underflow of destination exponent
54	OVF	Floating overflow of destination exponent
53	DZE	Floating divide with divisor of zero
52	INV	Floating invalid operand value
51:0	RAZ/IGN	Read as zero; ignored when written

A.5 Decodable Pseudo-Operations

Table A–5 lists the decodable pseudo-operations and their associated actual instructions.

Table A–5 Decodable Pseudo-Operations

Pseudo-Operation		Actual Instruction	
BR	target	BR	R31,target
CLR	Rx	BIS	R31,R31,Rx
FABS	Fx,Fy	CPYS	F31,Fx,Fy
FCLR	Fx	CPYS	F31,F31,Fx
FMOV	Fx,Fy	CPYS	Fx,Fx,Fy
FNEG	Fx,Fy	CPYSN	Fx,Fx,Fy
FNOP		CPYS	F31,F31,F31
MOV	Lit,Rx	LDA	Rx,lit(R31)
MOV	{Rx/Lit8},Ry	BIS	R31,{Rx/Lit8},Ry
MF_FPCR	Fx	MF_FPCR	Fx,Fx,Fx
MT_FPCR	Fx	MT_FPCR	Fx,Fx,Fx
NEGF	Fx,Fy	SUBF	F31,Fx,Fy
NEGF/S	Fx,Fy	SUBF/S	F31,Fx,Fy
NEGG	Fx,Fy	SUBG	F31,Fx,Fy
NEGG/S	Fx,Fy	SUBG/S	F31,Fx,Fy
NEGL	{Rx/Lit8},Ry	SUBL	R31,{Rx/Lit},Ry
NEGL/V	{Rx/Lit8},Ry	SUBL/V	R31,{Rx/Lit},Ry
NEGQ	{Rx/Lit8},Ry	SUBQ	R31,{Rx/Lit},Ry
NEGQ/V	{Rx/Lit8},Ry	SUBQ/V	R31,{Rx/Lit},Ry
NEGS	Fx,Fy	SUBS	F31,Fx,Fy
NEGS/SU	Fx,Fy	SUBS/SU	F31,Fx,Fy
NEGS/SUI	Fx,FY	SUBS/SUI	F31,Fx,Fy
NEGT	Fx,Fy	SUBT	F31,Fx,Fy
NEGT/SU	Fx,Fy	SUBT/SU	F31,Fx,Fy
NEGT/SUI	Fx,FY	SUBT/SUI	F31,Fx,Fy
NOP		BIS	R31,R31,R31
NOT	{Rx/Lit8},Ry	ORNOT	R31,{Rx/Lit},Ry
SEXTL	{Rx/Lit},Ry	ADDL	R31,{Rx/Lit},Ry
UNOP		LDQ_U	R31,0(Rx)

A.6 Common Architecture Opcodes in Numerical Order

Table A–6 lists the common architecture opcodes in numerical order

Table A–6 Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
00	CALL_PAL	11.26	CMOVNE	15.01E	CVTDG /C
01	OPC01	11.28	ORNOT	15.020	ADDG /C
02	OPC02	11.40	XOR	15.021	SUBG /C
03	OPC03	11.44	CMOVLT	15.022	MULG /C
04	OPC04	11.46	CMOVGE	15.023	DIVG /C
05	OPC05	11.48	EQV	15.02C	CVTGF /C
06	OPC06	11.64	CMOVLE	15.02D	CVTGD /C
07	OPC07	11.66	CMOVGT	15.02F	CVTGQ /C
08	LDA	12.02	MSKBL	15.03C	CVTQF /C
09	LDAH	12.06	EXTBL	15.03E	CVTQG /C
0A	OPC0A	12.0B	INSBL	15.080	ADDF
0B	LDQ_U	12.12	MSKWL	15.081	SUBF
0C	OPC0C	12.16	EXTWL	15.082	MULF
0D	OPC0D	12.1B	INSWL	15.083	DIVF
0E	OPC0E	12.22	MSKLL	15.09E	CVTDG
0F	STQ_U	12.26	EXTLL	15.0A0	ADDG
10.00	ADDL	12.2B	INSLL	15.0A1	SUBG
10.02	S4ADDL	12.30	ZAP	15.0A2	MULG
10.09	SUBL	12.31	ZAPNOT	15.0A3	DIVG
10.0B	S4SUBL	12.32	MSKQL	15.0A5	CMPGEQ
10.0F	CMPBGE	12.34	SRL	15.0A6	CMPGLT
10.12	S8ADDL	12.36	EXTQL	15.0A7	CMPGLE
10.1B	S8SUBL	12.39	SLL	15.0AC	CVTGF
10.1D	CMPULT	12.3B	INSQL	15.0AD	CVTGD
10.20	ADDQ	12.3C	SRA	15.0AF	CVTGQ
10.22	S4ADDQ	12.52	MSKWH	15.0BC	CVTQF
10.29	SUBQ	12.57	INSWH	15.0BE	CVTQG
10.2B	S4SUBQ	12.5A	EXTWH	15.100	ADDF /UC
10.2D	CMPEQ	12.62	MSKLH	15.101	SUBF /UC
10.32	S8ADDQ	12.67	INSLH	15.102	MULF /UC
10.3B	S8SUBQ	12.6A	EXTLH	15.103	DIVF /UC
10.3D	CMPULE	12.72	MSKQH	15.11E	CVTDG /UC
10.40	ADDL/V	12.77	INSQH	15.120	ADDG /UC
10.49	SUBL/V	12.7A	EXTQH	15.121	SUBG /UC
10.4D	CMPLT	13.00	MULL	15.122	MULG /UC
10.60	ADDQ/V	13.20	MULQ	15.123	DIVG /UC

(continued on next page)

Table A–6 (Cont.) Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
10.69	SUBQ/V	13.30	UMULH	15.12C	CVTGF /UC
10.6D	CMPLE	13.40	MULL/V	15.12D	CVTGD /UC
11.00	AND	13.60	MULQ/V	15.12F	CVTGQ /VC
11.08	BIC	14	OPC14	15.180	ADDF /U
11.14	CMOVLBS	15.000	ADDF/C	15.181	SUBF /U
11.16	CMOVLBC	15.001	SUBF/C	15.182	MULF /U
11.20	BIS	15.002	MULF/C	15.183	DIVF /U
11.24	CMOVEQ	15.003	DIVF/C	15.19E	CVTDG /U
15.1A0	ADDG/U	15.580	ADDF/SU	16.0A6	CMPTLT
15.1A1	SUBG/U	15.581	SUBF/SU	16.0A7	CMPTLE
15.1A2	MULG/U	15.582	MULF/SU	16.0AC	CVTTS
15.1A3	DIVG/U	15.583	DIVF/SU	16.0AF	CVTTQ
15.1AC	CVTGF/U	15.59E	CVTDG/SU	16.0BC	CVTQS
15.1AD	CVTGD/U	15.5A0	ADDG/SU	16.0BE	CVTQT
15.1AF	CVTGQ/V	15.5A1	SUBG/SU	16.0C0	ADDS /D
15.400	ADDF/SC	15.5A2	MULG/SU	16.0C1	SUBS /D
15.401	SUBF/SC	15.5A3	DIVG/SU	16.0C2	MULS /D
15.402	MULF/SC	15.5AC	CVTGF/SU	16.0C3	DIVS /D
15.403	DIVF/SC	15.5AD	CVTGD/SU	16.0E0	ADDT /D
15.41E	CVTDG/SC	15.5AF	CVTGQ/SV	16.0E1	SUBT /D
15.420	ADDG/SC	16.000	ADDS/C	16.0E2	MULT /D
15.421	SUBG/SC	16.001	SUBS/C	16.0E3	DIVT /D
15.422	MULG/SC	16.002	MULS/C	16.0EC	CVTTS /D
15.423	DIVG/SC	16.003	DIVS/C	16.0EF	CVTTQ /D
15.42C	CVTGF/SC	16.020	ADDT/C	16.0FC	CVTQS /D
15.42D	CVTGD/SC	16.021	SUBT/C	16.0FE	CVTQT /D
15.42F	CVTGQ/SC	16.022	MULT/C	16.100	ADDS /UC
15.480	ADDF/S	16.023	DIVT/C	16.101	SUBS /UC
15.481	SUBF/S	16.02C	CVTTS/C	16.102	MULS /UC
15.482	MULF/S	16.02F	CVTTQ/C	16.103	DIVS /UC
15.483	DIVF/S	16.03C	CVTQS/C	16.120	ADDT /UC

(continued on next page)

Table A–6 (Cont.) Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
15.49E	CVTDG/S	16.03E	CVTQT/C	16.121	SUBT /UC
15.4A0	ADDG/S	16.040	ADDS/M	16.122	MULT /UC
15.4A1	SUBG/S	16.041	SUBS/M	16.123	DIVT /UC
15.4A2	MULG/S	16.042	MULS/M	16.12C	CVTTS /UC
15.4A3	DIVG/S	16.043	DIVS/M	16.12F	CVTTQ /VC
15.4A5	CMPGEQ/S	16.060	ADDT/M	16.140	ADDS /UM
15.4A6	CMPGLT/S	16.061	SUBT/M	16.141	SUBS /UM
15.4A7	CMPGLE/S	16.062	MULT/M	16.142	MULS /UM
15.4AC	CVTGF/S	16.063	DIVT/M	16.143	DIVS /UM
15.4AD	CVTGD/S	16.06C	CVTTS/M	16.160	ADDT /UM
15.4AF	CVTGQ/S	16.06F	CVTTQ/M	16.161	SUBT /UM
15.500	ADDF/SUC	16.07C	CVTQS/M	16.162	MULT /UM
15.501	SUBF/SUC	16.07E	CVTQT/M	16.163	DIVT /UM
15.502	MULF/SUC	16.080	ADDS	16.16C	CVTTS /UM
15.503	DIVF/SUC	16.081	SUBS	16.16F	CVTTQ /VM
15.51E	CVTDG/SUC	16.082	MULS	16.180	ADDS /U
15.520	ADDG/SUC	16.083	DIVS	16.181	SUBS /U
15.521	SUBG/SUC	16.0A0	ADDT	16.182	MULS /U
15.522	MULG/SUC	16.0A1	SUBT	16.183	DIVS /U
15.523	DIVG/SUC	16.0A2	MULT	16.1A0	ADDT /U
15.52C	CVTGF/SUC	16.0A3	DIVT	16.1A1	SUBT /U
15.52D	CVTGD/SUC	16.0A4	CMPTUN	16.1A2	MULT /U
15.52F	CVTGQ/SVC	16.0A5	CMPTEQ	16.1A3	DIVT /U
16.1AC	CVTTS/U	16.5AF	CVTTQ/SV	16.7BC	CVTQS /SUI
16.1AF	CVTTQ/V	16.5C0	ADDS/SUD	16.7BE	CVTQT /SUI
16.1C0	ADDS/UD	16.5C1	SUBS/SUD	16.7C0	ADDS /SUID
16.1C1	SUBS/UD	16.5C2	MULS/SUD	16.7C1	SUBS /SUID
16.1C2	MULS/UD	16.5C3	DIVS/SUD	16.7C2	MULS /SUID

(continued on next page)

Table A–6 (Cont.) Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
16.1C3	DIVS/UD	16.5E0	ADDT/SUD	16.7C3	DIVS /SUID
16.1E0	ADDT/UD	16.5E1	SUBT/SUD	16.7E0	ADDT /SUID
16.1E1	SUBT/UD	16.5E2	MULT/SUD	16.7E1	SUBT /SUID
16.1E2	MULT/UD	16.5E3	DIVT/SUD	16.7E2	MULT /SUID
16.1E3	DIVT/UD	16.5EC	CVTTS/SUD	16.7E3	DIVT /SUID
16.1EC	CVTTS/UD	16.5EF	CVTTQ/SVD	16.7EC	CVTTS /SUID
16.1EF	CVTTQ/VD	16.6AC	CVTST/S	16.7EF	CVTTQ /SVID
16.2AC	CVTST	16.700	ADDS/SUIC	16.7FC	CVTQS /SUID
16.500	ADDS/SUC	16.701	SUBS/SUIC	16.7FE	CVTQT /SUID
16.501	SUBS/SUC	16.702	MULS/SUIC	17.010	CVTLQ
16.502	MULS/SUC	16.703	DIVS/SUIC	17.020	CPYS
16.503	DIVS/SUC	16.720	ADDT/SUIC	17.021	CPYSN
16.520	ADDT/SUC	16.721	SUBT/SUIC	17.022	CPYSE
16.521	SUBT/SUC	16.722	MULT/SUIC	17.024	MT_ FPCR
16.522	MULT/SUC	16.723	DIVT/SUIC	17.025	MF_ FPCR
16.523	DIVT/SUC	16.72C	CVTTS/SUIC	17.02A	FCMOVEQ
16.52C	CVTTS/SUC	16.72F	CVTTQ/SVIC	17.02B	FCMOVNE
16.52F	CVTTQ/SVC	16.73C	CVTQS/SUIC	17.02C	FCMOVLT
16.540	ADDS/SUM	16.73E	CVTQT/SUIC	17.02D	FCMOVGE
16.541	SUBS/SUM	16.740	ADDS/SUIM	17.02E	FCMOVLE
16.542	MULS/SUM	16.741	SUBS/SUIM	17.02F	FCMOVGT
16.543	DIVS/SUM	16.742	MULS/SUIM	17.030	CVTQL
16.560	ADDT/SUM	16.743	DIVS/SUIM	17.130	CVTQL /V
16.561	SUBT/SUM	16.760	ADDT/SUIM	17.530	CVTQL /SV
16.562	MULT/SUM	16.761	SUBT/SUIM	18.0000	TRAPB
16.563	DIVT/SUM	16.762	MULT/SUIM	18.0400	EXCB
16.56C	CVTTS/SUM	16.763	DIVT/SUIM	18.4000	MB
16.56F	CVTTQ/SVM	16.76C	CVTTS/SUIM	18.4400	WMB
16.580	ADDS/SU	16.76F	CVTTQ/SVIM	18.8000	FETCH
16.581	SUBS/SU	16.77C	CVTQS/SUIM	18.A000	FETCH_ M
16.582	MULS/SU	16.77E	CVTQT/SUIM	18.C000	RPCC
16.583	DIVS/SU	16.780	ADDS/SUI	18.E000	RC
16.5A0	ADDT/SU	16.781	SUBS/SUI	18.F000	RS
16.5A1	SUBT/SU	16.782	MULS/SUI	19	PAL19
16.5A2	MULT/SU	16.783	DIVS/SUI	1A.0	JMP
16.5A3	DIVT/SU	16.7A0	ADDT/SUI	1A.1	JSR
16.5A4	CMPTUN/SU	16.7A1	SUBT/SUI	1A.2	RET
16.5A5	CMPTEQ/SU	16.7A2	MULT/SUI	1A.3	JSR_ COROUTINE
16.5A6	CMPTLT/SU	16.7A3	DIVT/SUI	1B	PAL1B

(continued on next page)

Table A–6 (Cont.) Common Architecture Opcodes in Numerical Order

Opcode		Opcode		Opcode	
16.5A7	CMPTLE/SU	16.7AC	CVTTS/SUI	1C	OPC1C
16.5AC	CVTTS/SU	16.7AF	CVTTQ/SVI	1D	PAL1D
1E	PAL1E	2A	LDL_L	36	FBGE
1F	PAL1F	2B	LDQ_L	37	FBGT
20	LDF	2C	STL	38	BLBC
21	LDG	2D	STQ	39	BEQ
22	LDS	2E	STL_C	3A	BLT
23	LDT	2F	STQ_C	3B	BLE
24	STF	30	BR	3C	BLBS
25	STG	31	FBEQ	3D	BNE
26	STS	32	FBLT	3E	BGE
27	STT	33	FBLE	3F	BGT
28	LDL	34	BSR		
29	LDQ	35	FBNE		

A.7 OpenVMS PALcode Instruction Summary

Table A–7 lists the OpenVMS unprivileged PALcode instructions and Table A–8 lists the OpenVMS privileged PALcode instructions.

Table A–7 OpenVMS Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
AMOVRM	00.00A1	Atomic move from register to memory
AMOVRR	00.00A0	Atomic move from register to register
BPT	00.0080	Breakpoint
BUGCHK	00.0081	Bugcheck
CHMK	00.0083	Change mode to kernel
CHME	00.0082	Change mode to executive
CHMS	00.0084	Change mode to supervisor
CHMU	00.0085	Change mode to user
GENTRAP	00.00AA	Generate software trap
IMB	00.0086	I-stream memory barrier
INSQHIL	00.0087	Insert into longword queue at head interlocked
INSQHILR	00.00A2	Insert into longword queue at head interlocked resident
INSQHIQ	00.0089	Insert into quadword queue at head interlocked
INSQHIQR	00.00A4	Insert into quadword queue at head interlocked resident
INSQTIL	00.0088	Insert into longword queue at tail interlocked
INSQTILR	00.00A3	Insert into longword queue at tail interlocked resident
INSQTIQ	00.008A	Insert into quadword queue at tail interlocked
INSQTIQR	00.00A5	Insert into quadword queue at tail interlocked resident
INSQUEL	00.008B	Insert entry into longword queue
INSQUEL/D	00.008D	Insert entry into longword queue deferred
INSQUEQ	00.008C	Insert entry into quadword queue
INSQUEQ/D	00.008E	Insert entry into quadword queue deferred
PROBER	00.008F	Probe for read access
PROBEW	00.0090	Probe for write access
RD_PS	00.0091	Move processor status
READ_UNQ	00.009E	Read unique context
REI	00.0092	Return from exception or interrupt

(continued on next page)

Table A–7 (Cont.) OpenVMS Unprivileged PALcode Instructions

Mnemonic	Opcode	Description
REMQHIL	00.0093	Remove from longword queue at head interlocked
REMQHILR	00.00A6	Remove from longword queue at head interlocked resident
REMQHIQ	00.0095	Remove from quadword queue at head interlocked
REMQHIQR	00.00A8	Remove from quadword queue at head interlocked resident
REMQTIL	00.0094	Remove from longword queue at tail interlocked
REMQTILR	00.00A7	Remove from longword queue at tail interlocked resident
REMQTIQ	00.0096	Remove from quadword queue at tail interlocked
REMQTIQR	00.00A9	Remove from quadword queue at tail interlocked resident
REMQUEL	00.0097	Remove entry from longword queue
REMQUEL/D	00.0099	Remove entry from longword queue deferred
REMQUEQ	00.0098	Remove entry from quadword queue
REMQUEQ/D	00.009A	Remove entry from quadword queue deferred
RSCC	00.009D	Read system cycle counter
SWASTEN	00.009B	Swap AST enable for current mode
WRITE_UNQ	00.009F	Write unique context
WR_PS_SW	00.009C	Write processor status software field

Table A–8 OpenVMS Privileged PALcode Instructions

Mnemonic	Opcode	Description
CFLUSH	00.0001	Cache flush
CSERVE	00.0009	Console service
DRAINA	00.0002	Drain aborts
HALT	00.0000	Halt processor
LDQP	00.0003	Load quadword physical
MFPR_ASN	00.0006	Move from processor register ASN
MFPR_ESP	00.001E	Move from processor register ESP
MFPR_FEN	00.000B	Move from processor register FEN
MFPR_IPL	00.000E	Move from processor register IPL
MFPR_MCES	00.0010	Move from processor register MCES
MFPR_PCBB	00.0012	Move from processor register PCBB
MFPR_PRBR	00.0013	Move from processor register PRBR
MFPR_PTBR	00.0015	Move from processor register PTBR
MFPR_SCBB	00.0016	Move from processor register SCBB
MFPR_SISR	00.0019	Move from processor register SISR
MFPR_SSP	00.0020	Move from processor register SSP
MFPR_TBCHK	00.001A	Move from processor register TBCHK
MFPR_USP	00.0022	Move from processor register USP
MFPR_VPTB	00.0029	Move from processor register VPTB
MFPR_WHAMI	00.003F	Move from processor register WHAMI
MTPR_ASTEN	00.0026	Move to processor register ASTEN
MTPR_ASTSR	00.0027	Move to processor register ASTSR
MTPR_DATFX	00.002E	Move to processor register DATFX
MTPR_ESP	00.001F	Move to processor register ESP
MTPR_FEN	00.000B	Move to processor register FEN
MTPR_IPIR	00.000D	Move to processor register IPRI
MTPR_IPL	00.000E	Move to processor register IPL
MTPR_MCES	00.0011	Move to processor register MCES
MTPR_PERFMON	00.002B	Move to processor register PERFMON
MTPR_PRBR	00.0014	Move to processor register PRBR

(continued on next page)

Table A–8 (Cont.) OpenVMS Privileged PALcode Instructions

Mnemonic	Opcode	Description
MTPR_SCBB	00.0017	Move to processor register SCBB
MTPR_SIRR	00.0018	Move to processor register SIRR
MTPR_SSP	00.0021	Move to processor register SSP
MTPR_TBIA	00.001B	Move to processor register TBIA
MTPR_TBIAP	00.001C	Move to processor register TBIAP
MTPR_TBIS	00.001D	Move to processor register TBIS
MTPR_TBISD	00.0024	Move to processor register TBISD
MTPR_TBISI	00.0025	Move to processor register TBISI
MTPR_USP	00.0023	Move to processor register USP
MTPR_VPTB	00.002A	Move to processor register VPTB
STQP	00.0004	Store quadword physical
SWPCTX	00.0005	Swap privileged context
SWPPAL	00.000A	Swap PALcode image

A.8 PALcode Opcodes in Numerical Order

Table A–9 lists the PALcode opcodes in numerical order.

Table A–9 PALcode Opcodes in Numerical Order

$Opcode_{16}$	$Opcode_{10}$	OpenVMS
00.0000	00.0000	HALT
00.0001	00.0001	CFLUSH
00.0002	00.0002	DRAINA
00.0003	00.0003	LDQP
00.0004	00.0004	STQP
00.0005	00.0005	SWPCTX
00.0006	00.0006	MFPR_ASN
00.0007	00.0007	MTPR_ASTEN
00.0008	00.0008	MTPR_ASTSR
00.0009	00.0009	CSERVE
00.000A	00.0010	SWPPAL
00.000B	00.0011	MFPR_FEN
00.000C	00.0012	MTPR_FEN
00.000D	00.0013	MTPR_IPIR
00.000E	00.0014	MFPR_IPL
00.000F	00.0015	MTPR_IPL
00.0010	00.0016	MFPR_MCES
00.0011	00.0017	MTPR_MCES
00.0012	00.0018	MFPR_PCBB
00.0013	00.0019	MFPR_PRBR
00.0014	00.0020	MTPR_PRBR
00.0015	00.0021	MFPR_PTBR
00.0016	00.0022	MFPR_SCBB
00.0017	00.0023	MTPR_SCBB
00.0018	00.0024	MTPR_SIRR
00.0019	00.0025	MFPR_SISR
00.001A	00.0026	MFPR_TBCHK
00.001B	00.0027	MTPR_TBIA
00.001C	00.0028	MTPR_TBIAP

(continued on next page)

Table A–9 (Cont.) PALcode Opcodes in Numerical Order

$Opcode_{16}$	$Opcode_{10}$	OpenVMS
00.001D	00.0029	MTPR_TBIS
00.001E	00.0030	MFPR_ESP
00.001F	00.0031	MTPR_ESP
00.0020	00.0032	MFPR_SSP
00.0021	00.0033	MTPR_SSP
00.0022	00.0034	MFPR_USP
00.0023	00.0035	MTPR_USP
00.0024	00.0036	MTPR_TBISD
00.0025	00.0037	MTPR_TBISI
00.0026	00.0038	MFPR_ASTEN
00.0027	00.0039	MFPR_ASTSR
00.0029	00.0040	MFPR_VPTB
00.002A	00.0041	MTPR_VPTB
00.002B	00.0042	MTPR_PERFMON
00.002E	00.0043	MTPR_DATFX
00.003F	00.0063	MFPR_WHAMI
00.0080	00.0128	BPT
00.0081	00.0129	BUGCHK
00.0082	00.0130	CHME
00.0083	00.0131	CHMK
00.0084	00.0132	CHMS
00.0085	00.0133	CHMU
00.0086	00.0134	IMB
00.0087	00.0135	INSQHIL
00.0088	00.0136	INSQTIL
00.0089	00.0137	INSQHIQ
00.008A	00.0138	INSQTIQ
00.008B	00.0139	INSQUEL
00.008C	00.0140	INSQUEQ
00.008D	00.0141	INSQUEL/D
00.008E	00.0142	INSQUEQ/D
00.008F	00.0143	PROBER
00.0090	00.0144	PROBEW
00.0091	00.0145	RD_PS
00.0092	00.0146	REI
00.0093	00.0147	REMQHIL
00.0094	00.0148	REMQTIL
00.0095	00.0149	REMQHIQ
00.0096	00.0150	REMQTIQ
00.0097	00.0151	REMQUEL
00.0098	00.0152	REMQUEQ
00.0099	00.0153	REMQUEL/D
00.009A	00.0154	REMQUEQ/D
00.009B	00.0155	SWASTEN
00.009C	00.0156	WR_PS_SW
00.009D	00.0157	RSCC
00.009E	00.0158	READ_UNQ
00.009F	00.0159	WRITE_UNQ
00.00A0	00.0160	AMOVRR
00.00A1	00.0161	AMOVRM
00.00A2	00.0162	INSQHILR
00.00A3	00.0163	INSQTILR

(continued on next page)

Table A–9 (Cont.) PALcode Opcodes in Numerical Order

$Opcode_{16}$	$Opcode_{10}$	OpenVMS
00.00A4	00.0164	INSQHIQR
00.00A5	00.0165	INSQTIQR
00.00A6	00.0166	REMQHILR
00.00A7	00.0167	REMQTILR
00.00A8	00.0168	REMQHIQR
00.00A9	00.0169	REMQTIQR
00.00AA	00.0170	GENTRAP

A.9 Common Architecture Instructions

Table A–10 lists the common architecture instructions. Where enclosed in square brackets, Ra.wq defaults to R31. This occurs for instructions BR, JMP, and RET.

Table A–10 Common Architecture Instructions

Mnemonic	Code	Operands	Operation
ADDF	15.080	Fa.rf,Fb.rf,Fc.wf	Fc←Fav + Fbv
ADDF/C	15.000		
ADDF/S	15.480		
ADDF/SC	15.400		
ADDF/SUC	15.500		
ADDF/SU	15.580		
ADDF/U	15.180		
ADDF/UC	15.100		
ADDG	15.0A0	Fa.rg,Fb.rg,Fc.wg	Fc←Fav + Fbv
ADDG/C	15.020		
ADDG/S	15.4A0		
ADDG/SC	15.420		
ADDG/SU	15.5A0		
ADDG/SUC	15.520		
ADDG/U	15.1A0		
ADDG/UC	15.120		
ADDL	10.00	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT((Rav + Rbv)<31:0>)
ADDL/V	10.40		
ADDQ	10.20	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav + Rbv
ADDQ/V	10.60		
ADDS	16.080	Fa.rs,Fb.rs,Fc.ws	Fc←Fav + Fbv
ADDS/C	16.000		
ADDS/D	16.0C0		
ADDS/M	16.040		
ADDS/SU	16.580		
ADDS/SUC	16.500		
ADDS/SUD	16.5C0		
ADDS/SUI	16.780		
ADDS/SUIC	16.700		
ADDS/SUID	16.7C0		
ADDS/SUIM	16.740		
ADDS/SUM	16.540		
ADDS/U	16.180		
ADDS/UC	16.100		

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
ADDS/UD	16.1C0		
ADDS/UM	16.140		
ADDT	16.0A0	Fa.rt,Fb.rt,Fc.wt	Fc←Fav + Fbv
ADDT/C	16.020		
ADDT/D	16.0E0		
ADDT/M	16.060		
ADDT/SU	16.5A0		
ADDT/SUC	16.520		
ADDT/SUD	16.5E0		
ADDT/SUI	16.7A0		
ADDT/SUIC	16.720		
ADDT/SUID	16.7E0		
ADDT/SUIM	16.760		
ADDT/SUM	16.560		
ADDT/U	16.1A0		
ADDT/UC	16.120		
ADDT/UD	16.1E0		
ADDT/UM	16.160		
AND	11.00	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav AND Rbv
BEQ	39	Ra.rq,disp.al	If Rav = 0 Then PC←PC + {4*SEXT(disp)}
BGE	3E	Ra.rq,disp.al	If Rav ≥ 0 Then PC←PC + {4*SEXT(disp)}
BGT	3F	Ra.rq,disp.al	If Rav > 0 Then PC←PC + {4*SEXT(disp)}
BIC	11.08	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav AND {NOT Rbv}
BIS	11.20	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav OR Rbv
BLBC	38	Ra.rq,disp.al	If Rav<0> = 0 Then PC←PC + {4*SEXT(disp)}
BLBS	3C	Ra.rq,disp.al	If Rav<0> = 1 Then PC←PC + {4*SEXT(disp)}
BLE	3B	Ra.rq,disp.al	If Rav ≤ 0 Then PC←PC + {4*SEXT(disp)}
BLT	3A	Ra.rq,disp.al	If Rav < 0 Then PC←PC + {4*SEXT(disp)}
BNE	3D	Ra.rq,disp.al	If Rav ≠ 0 Then PC←PC + {4*SEXT(disp)}
BR	30	[Ra.wq],disp.al	Ra←PC; PC←PC + {4*SEXT(disp)}
BSR	34	Ra.wq,disp.al	Ra←PC; PC←PC + {4*SEXT(disp)}
CALL_PAL	00	fnc.ir	Trap to PALcode
CMOVEQ	11.24	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav = 0 Then Rc←Rbv
CMOVGE	11.46	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav ≥ 0 Then Rc←Rbv
CMOVGT	11.66	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav > 0 Then Rc←Rbv
CMOVLBC	11.16	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav<0> = 0 Then Rc←Rbv
CMOVLBS	11.14	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav<0> = 1 Then Rc←Rbv
CMOVLE	11.64	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav ≤ 0 Then Rc←Rbv
CMOVLT	11.44	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav < 0 Then Rc←Rbv
CMOVNE	11.26	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav ≠ 0 Then Rc←Rbv
CMPBGE	10.0F	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Bytewise compare mask of {Rav ≥ Rbv}
CMPEQ	10.2D	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav = Rbv Then Rc←1 Else Rc←0
CMPGEQ	15.0A5	Fa.rg,Fb.rg,Fc.wq	If Fav = Fbv Then Fc←0.5 Else Fc←0.0
CMPGEQ/S	15.4A5		
CMPGLE	15.0A7	Fa.rg,Fb.rg,Fc.wq	If Fav ≤ Fbv Then Fc←0.5 Else Fc←0.0
CMPGLE/S	15.4A7		
CMPGLT	15.0A6	Fa.rg,Fb.rg,Fc.wq	If Fav < Fbv Then Fc←0.5 Else Fc←0.0
CMPGLT/S	15.4A6		
CMPLE	10.6D	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav ≤ Rbv Then Rc←1 Else Rc←0
CMPLT	10.4D	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav < Rbv Then Rc←1 Else Rc←0

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
CMPTEQ	16.0A5	Fa.rt,Fb.rt,Fc.wq	If Fav = Fbv Then Fc ←2.0 Else Fc←0.0
CMPTEQ/SU	16.5A5		
CMPTLE	16.0A7	Fa.rt,Fb.rt,Fc.wq	If Fav ≤ Fbv Then Fc←2.0 Else Fc←0.0
CMPTLE/SU	16.5A7		
CMPTLT	16.0A6	Fa.rt,Fb.rt,Fc.wq	If Fav < Fbv Then Fc←2.0 Else Fc←0.0
CMPTLT/SU	16.5A6		
CMPTUN	16.0A4	Fa.rt,Fb.rt,Fc.wq	If Fav Unordered Fbv Then Fc←2.0 Else Fc←0.0
CMPTUN/SU	16.5A4		
CMPULE	10.3D	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav U≤ Rbv Then Rc←1 Else Rc←0
CMPULT	10.1D	Ra.rq,{Rb.rq/#b.ib},Rc.wq	If Rav U< Rbv Then Rc←1 Else Rc←0
CPYS	17.020	Fa.rq,Fb.rq,Fc.wq	Fc←Fav<63> || Fbv<62:0>
CPYSE	17.022	Fa.rq,Fb.rq,Fc.wq	Fc←Fav<63:52> || Fbv<51:0>
CPYSN	17.021	Fa.rq,Fb.rq,Fc.wq	Fc←NOT Fav<63> || Fbv<62:0>
CVTDG	15.09E	Fb.rd,Fc.wg	Fc←D_Float to G_Float of Fbv
CVTDG/C	15.01E		
CVTDG/S	15.49E		
CVTDG/SC	15.41E		
CVTDG/SU	15.59E		
CVTDG/SUC	15.51E		
CVTDG/U	15.19E		
CVTDG/UC	15.11E		
CVTGD	15.0AD	Fb.rg,Fc.wd	Fc←G_Float to D_Float of Fbv
CVTGD/C	15.02D		
CVTGD/S	15.4AD		
CVTGD/SC	15.42D		
CVTGD/SU	15.5AD		
CVTGD/SUC	15.52D		
CVTGD/U	15.1AD		
CVTGD/UC	15.12D		
CVTGF	15.0AC	Fb.rg,Fc.wf	Fc←G_Float to F_Float of Fbv
CVTGF/C	15.02C		
CVTGF/S	15.4AC		
CVTGF/SC	15.42C		
CVTGF/SU	15.5AC		
CVTGF/SUC	15.52C		
CVTGF/U	15.1AC		
CVTGF/UC	15.12C		
CVTGQ	15.0AF	Fb.rg,Fc.wq	Fc←G_Float to quad of Fbv
CVTGQ/C	15.02F		
CVTGQ/S	15.4AF		
CVTGQ/SC	15.42F		
CVTGQ/SV	15.5AF		
CVTGQ/SVC	15.52F		
CVTGQ/V	15.1AF		
CVTGQ/VC	15.12F		
CVTLQ	17.010	Fb.rl,Fc.wq	Fc←SEXT(Fbv<63:62>||Fbv<58:29>)
CVTQF	15.0BC	Fb.rq,Fc.wf	Fc←Quad to F_Float of Fbv
CVTQF/C	15.03C		
CVTQG	15.0BE	Fb.rq,Fc.wg	Fc←Quad to G_Float of Fbv
CVTQG/C	15.03E		
CVTQL	17.030	Fb.rq,Fc.wl	Fc←Quad to long of Fbv

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
CVTQL/SV	17.530		
CVTQL/V	17.130		
CVTQS	16.0BC	Fb.rq,Fc.ws	Fc ←Quad to S_Float of Fbv
CVTQS/C	16.03C		
CVTQS/D	16.0FC		
CVTQS/M	16.07C		
CVTQS/SUI	16.7BC		
CVTQS/SUIC	16.73C		
CVTQS/SUID	16.7FC		
CVTQS/SUIM	16.77C		
CVTQT	16.0BE	Fb.rq,Fc.wt	Fc←Quad to T_Float of Fbv
CVTQT/C	16.03E		
CVTQT/D	16.0FE		
CVTQT/M	16.07E		
CVTQT/SUI	16.7BE		
CVTQT/SUIC	16.73E		
CVTQT/SUID	16.7FE		
CVTQT/SUIM	16.77E		
CVTST	16.2AC	Fb.rs,Fc.wt	Fc←S_Float to T_Float of Fbv
CVTST/S	16.6AC		
CVTTQ	16.0AF	Fb.rt,Fc.wq	Fc←T_Float to quad of Fbv
CVTTQ/C	16.02F		
CVTTQ/D	16.0EF		
CVTTQ/M	16.06F		
CVTTQ/SV	16.5AF		
CVTTQ/SVC	16.52F		
CVTTQ/SVD	16.5EF		
CVTTQ/SVI	16.7AF		
CVTTQ/SVIC	16.72F		
CVTTQ/SVID	16.7EF		
CVTTQ/SVIM	16.76F		
CVTTQ/SVM	16.56F		
CVTTQ/V	16.1AF		
CVTTQ/VC	16.12F		
CVTTQ/VD	16.1EF		
CVTTQ/VM	16.16F		
CVTTS	16.0AC	Fb.rt,Fc.ws	Fc←T_Float to S_Float of Fbv
CVTTS/C	16.02C		
CVTTS/D	16.0EC		
CVTTS/M	16.06C		
CVTTS/SU	16.5AC		
CVTTS/SUC	16.52C		
CVTTS/SUD	16.5EC		
CVTTS/SUI	16.7AC		
CVTTS/SUIC	16.72C		
CVTTS/SUID	16.7EC		
CVTTS/SUIM	16.76C		
CVTTS/SUM	16.56C		
CVTTS/U	16.1AC		
CVTTS/UC	16.12C		
CVTTS/UD	16.1EC		
CVTTS/UM	16.16C		

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
DIVF	15.083	Fa.rf,Fb.rf,Fc.wf	Fc←Fav / Fbv
DIVF/C	15.003		
DIVF/S	15.483		
DIVF/SC	15.403		
DIVF/SU	15.583		
DIVF/SUC	15.503		
DIVF/U	15.183		
DIVF/UC	15.103		
DIVG	15.0A3	Fa.rg,Fb.rg,Fc.wg	Fc←Fav / Fbv
DIVG/C	15.023		
DIVG/S	15.4A3		
DIVG/SC	15.423		
DIVG/SU	15.5A3		
DIVG/SUC	15.523		
DIVG/U	15.1A3		
DIVG/UC	15.123		
DIVS	16.083	Fa.rs,Fb.rs,Fc.ws	Fc←Fav / Fbv
DIVS/C	16.003		
DIVS/D	16.0C3		
DIVS/M	16.043		
DIVS/SU	16.583		
DIVS/SUC	16.503		
DIVS/SUD	16.5C3		
DIVS/SUI	16.783		
DIVS/SUIC	16.703		
DIVS/SUID	16.7C3		
DIVS/SUIM	16.743		
DIVS/SUM	16.543		
DIVS/U	16.183		
DIVS/UC	16.103		
DIVS/UD	16.1C3		
DIVS/UM	16.143		
DIVT	16.0A3	Fa.rt,Fb.rt,Fc.wt	Fc←Fav / Fbv
DIVT/C	16.023		
DIVT/D	16.0E3		
DIVT/M	16.063		
DIVT/SU	16.5A3		
DIVT/SUC	16.523		
DIVT/SUD	16.5E3		
DIVT/SUI	16.7A3		
DIVT/SUIC	16.723		
DIVT/SUID	16.7E3		
DIVT/SUIM	16.763		
DIVT/SUM	16.563		
DIVT/U	16.1A3		
DIVT/UC	16.123		
DIVT/UD	16.1E3		
DIVT/UM	16.163		
EQV	11.48	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav XOR {NOT Rbv}
EXCB	18.0400		Exception barrier
EXTBL	12.06	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Byte extract low from Rav at Rbv<2:0>
EXTLH	12.6A	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Long extract high from Rav at Rbv<2:0>

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
EXTLL	12.26	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Long extract low from Rav at Rbv<2:0>
EXTQH	12.7A	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Quad extract high from Rav at Rbv<2:0>
EXTQL	12.36	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Quad extract low from Rav at Rbv<2:0>
EXTWH	12.5A	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Word extract high from Rav at Rbv<2:0>
EXTWL	12.16	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Word extract low from Rav at Rbv<2:0>
FBEQ	31	Fa.rq,disp.al	If Fav = 0 Then PC←PC + {4*SEXT(disp)}
FBGE	36	Fa.rq,disp.al	If Fav ≥ 0 Then PC←PC + {4*SEXT(disp)}
FBGT	37	Fa.rq,disp.al	If Fav > 0 Then PC←PC + {4*SEXT(disp)}
FBLE	33	Fa.rq,disp.al	If Fav ≤ 0 Then PC←PC + {4*SEXT(disp)}
FBLT	32	Fa.rq,disp.al	If Fav < 0 Then PC←PC + {4*SEXT(disp)}
FBNE	35	Fa.rq,disp.al	If Fav ≠ 0 Then PC←PC + {4*SEXT(disp)}
FCMOVEQ	17.02A	Fa.rq,Fb.rq,Fc.wq	If Fav = 0 Then Fc←Fbv
FCMOVGE	17.02D	Fa.rq,Fb.rq,Fc.wq	If Fav ≥ 0 Then Fc←Fbv
FCMOVGT	17.02F	Fa.rq,Fb.rq,Fc.wq	If Fav > 0 Then Fc←Fbv
FCMOVLE	17.02E	Fa.rq,Fb.rq,Fc.wq	If Fav ≤ 0 Then Fc←Fbv
FCMOVLT	17.02C	Fa.rq,Fb.rq,Fc.wq	If Fav < 0 Then Fc←Fbv
FCMOVNE	17.02B	Fa.rq,Fb.rq,Fc.wq	If Fav ≠ 0 Then Fc←Fbv
FETCH	18.8000	0(Rb.ab)	Prefetch around (Rbv)
FETCH_M	18.A000	0(Rb.ab)	Prefetch around (Rbv), modify intent
INSBL	12.0B	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Byte insert low from Rav at Rbv<2:0>
INSLH	12.67	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Long insert high from Rav at Rbv<2:0>
INSLL	12.2B	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Long insert low from Rav at Rbv<2:0>
INSQH	12.77	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Quad insert high from Rav at Rbv<2:0>
INSQL	12.3B	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Quad insert low from Rav at Rbv<2:0>
INSWH	12.57	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Word insert high from Rav at Rbv<2:0>
INSWL	12.1B	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Word insert low from Rav at Rbv<2:0>
JMP	1A.0	[Ra.wq],(Rb.ab),hint	Ra←PC; PC←Rbv AND {NOT 3}
JSR	1A.1	Ra.wq,(Rb.ab),hint	Ra←PC; PC←Rbv AND {NOT 3}
JSR_COROUTINE	1A.3	Ra.wq,(Rb.ab),hint	Ra←PC; PC←Rbv AND {NOT 3}
LDA	08	Ra.wq,disp.ab(Rb.ab)	Ra←Rbv + SEXT(disp)
LDAH	09	Ra.wq,disp.ab(Rb.ab)	Ra←Rbv + SEXT(disp*65536)
LDF	20	Fa.wf,disp.ab(Rb.ab)	Fa←({Rbv + SEXT(disp)})
LDG	21	Fa.wg,disp.ab(Rb.ab)	Fa←({Rbv + SEXT(disp)})
LDL	28	Ra.wq,disp.ab(Rb.ab)	Ra←SEXT(({Rbv + SEXT(disp)})<31:0>)
LDL_L	2A	Ra.wq,disp.ab(Rb.ab)	Ra←SEXT(({Rbv + SEXT(disp)})<31:0>)
LDQ	29	Ra.wq,disp.ab(Rb.ab)	Ra←({Rbv + SEXT(disp)})
LDQ_L	2B	Ra.wq,disp.ab(Rb.ab)	Ra←({Rbv + SEXT(disp)})
LDQ_U	0B	Ra.wq,disp.ab(Rb.ab)	Ra←({{Rbv + SEXT(disp)} AND NOT 7})
LDS	22	Fa.ws,disp.ab(Rb.ab)	Fa←({Rbv + SEXT(disp)})
LDT	23	Fa.wt,disp.ab(Rb.ab)	Fa←({Rbv + SEXT(disp)})
MB	18.4000		Memory barrier
MF_FPCR	17.025	Fa.rq,Fa.rq,Fa.wq	Fa←FPCR
MSKBL	12.02	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Byte mask low from Rav at Rbv<2:0>
MSKLH	12.62	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Long mask high from Rav at Rbv<2:0>
MSKLL	12.22	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Long mask low from Rav at Rbv<2:0>
MSKQH	12.72	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Quad mask high from Rav at Rbv<2:0>
MSKQL	12.32	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Quad mask low from Rav at Rbv<2:0>
MSKWH	12.52	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Word mask high from Rav at Rbv<2:0>
MSKWL	12.12	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Word mask low from Rav at Rbv<2:0>
MT_FPCR	17.024	Fa.rq,Fa.rq,Fa.wq	FPCR←Fa
MULF	15.082	Fa.rf,Fb.rf,Fc.wf	Fc←Fav * Fbv
MULF/C	15.002		

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
MULF/S	15.482		
MULF/SC	15.402		
MULF/SU	15.582		
MULF/SUC	15.502		
MULF/U	15.182		
MULF/UC	15.102		
MULG	15.0A2	Fa.rg,Fb.rg,Fc.wg	Fc←Fav ∗ Fbv
MULG/C	15.022		
MULG/S	15.4A2		
MULG/SC	15.422		
MULG/SU	15.5A2		
MULG/SUC	15.522		
MULG/U	15.1A2		
MULG/UC	15.122		
MULL	13.00	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT((Rav ∗ Rbv)<31:0>)
MULL/V	13.40		
MULQ	13.20	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav ∗ Rbv
MULQ/V	13.60		
MULS	16.082	Fa.rs,Fb.rs,Fc.ws	Fc←Fav ∗ Fbv
MULS/C	16.002		
MULS/D	16.0C2		
MULS/M	16.042		
MULS/SU	16.582		
MULS/SUC	16.502		
MULS/SUD	16.5C2		
MULS/SUI	16.782		
MULS/SUIC	16.702		
MULS/SUID	16.7C2		
MULS/SUIM	16.742		
MULS/SUM	16.542		
MULS/U	16.182		
MULS/UC	16.102		
MULS/UD	16.1C2		
MULS/UM	16.142		
MULT	16.0A2	Fa.rt,Fb.rt,Fc.wt	Fc←Fav ∗ Fbv
MULT/C	16.022		
MULT/D	16.0E2		
MULT/M	16.062		
MULT/SU	16.5A2		
MULT/SUC	16.522		
MULT/SUD	16.5E2		
MULT/SUI	16.7A2		
MULT/SUIC	16.722		
MULT/SUID	16.7E2		
MULT/SUIM	16.762		
MULT/SUM	16.562		
MULT/U	16.1A2		
MULT/UC	16.122		
MULT/UD	16.1E2		
MULT/UM	16.162		
ORNOT	11.28	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav OR {NOT Rbv}
RC	18.E000	Ra.wq	Ra←intr_flag; intr_flag←0

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
RET	1A.2	[Ra.wq],(Rb.ab),hint	Ra ←PC; PC←Rbv AND {NOT 3}
RPCC	18.C000	Ra.wq	Ra←Process cycle counter
RS	18.F000	Ra.wq	Ra←intr_flag; intr_flag←1
S4ADDL	10.02	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT(((SLL(Rav,2)) + Rbv)<31:0>)
S4ADDQ	10.22	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SLL(Rav,2) + Rbv
S4SUBL	10.0B	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT(((SLL(Rav,2)) − Rbv)<31:0>)
S4SUBQ	10.2B	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SLL(Rav,2) − Rbv
S8ADDL	10.12	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT(((SLL(Rav,3)) + Rbv)<31:0>)
S8ADDQ	10.32	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SLL(Rav,3) + Rbv
S8SUBL	10.1B	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT(((SLL(Rav,3)) − Rbv)<31:0>)
S8SUBQ	10.3B	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SLL(Rav,3) − Rbv
SLL	12.39	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SLL(Rav,Rvb<5:0>)
SRA	12.3C	Ra.rb,{Rb.rq/#b.ib},Rc.wq	Rc←SRA(Rav,Rvb<5:0>)
SRL	12.34	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←SRL(Rav,Rvb<5:0>)
STF	24	Fa.rf,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Fav
STG	25	Fa.rg,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Fav
STL	2C	Ra.rl,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})<31:0>←Rav<31:0>
STL_C	2E	Ra.ml,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})<31:0>←Rav<31:0>
STQ	2D	Ra.rq,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Rav
STQ_C	2F	Ra.mq,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Rav
STQ_U	0F	Ra.rq,disp.ab(Rb.ab)	({{Rbv + SEXT(disp)} AND NOT 7})←Rav
STS	26	Fa.rs,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Fav
STT	27	Fa.rt,disp.ab(Rb.ab)	({Rbv + SEXT(disp)})←Fav
SUBF	15.081	Fa.rf,Fb.rf,Fc.wf	Fc←Fav − Fbv
SUBF/C	15.001		
SUBF/S	15.481		
SUBF/SC	15.401		
SUBF/SU	15.581		
SUBF/SUC	15.501		
SUBF/U	15.181		
SUBF/UC	15.101		
SUBG	15.0A1	Fa.rg,Fb.rg,Fc.wg	Fc←Fav − Fbv
SUBG/C	15.021		
SUBG/S	15.4A1		
SUBG/SC	15.421		
SUBG/SU	15.5A1		
SUBG/SUC	15.521		
SUBG/U	15.1A1		
SUBG/UC	15.121		
SUBL	10.09	Ra.rl,{Rb.rl/#b.ib},Rc.wq	Rc←SEXT((Rav − Rbv)<31:0>)
SUBL/V	10.49		
SUBQ	10.29	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav − Rbv
SUBQ/V	10.69		
SUBS	16.081	Fa.rs,Fb.rs,Fc.ws	Fc←Fav − Fbv
SUBS/C	16.001		
SUBS/D	16.0C1		
SUBS/M	16.041		
SUBS/SU	16.581		
SUBS/SUC	16.501		
SUBS/SUD	16.5C1		
SUBS/SUI	16.781		
SUBS/SUIC	16.701		

(continued on next page)

Table A–10 (Cont.) Common Architecture Instructions

Mnemonic	Code	Operands	Operation
SUBS/SUID	16.7C1		
SUBS/SUIM	16.741		
SUBS/SUM	16.541		
SUBS/U	16.181		
SUBS/UC	16.101		
SUBS/UD	16.1C1		
SUBS/UM	16.141		
SUBT	16.0A1	Fa.rt,Fb.rt,Fc.wt	Fc ←Fav – Fbv
SUBT/C	16.021		
SUBT/D	16.0E1		
SUBT/M	16.061		
SUBT/SU	16.5A1		
SUBT/SUC	16.521		
SUBT/SUD	16.5E1		
SUBT/SUI	16.7A1		
SUBT/SUIC	16.721		
SUBT/SUID	16.7E1		
SUBT/SUIM	16.761		
SUBT/SUM	16.561		
SUBT/U	16.1A1		
SUBT/UC	16.121		
SUBT/UD	16.1E1		
SUBT/UM	16.161		
TRAPB	18.0000		Drain any pending traps
UMULH	13.30	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←{Rav *U Rbv}<127:64>
WMB	18.4400		Write memory barrier
XOR	11.40	Ra.rq,{Rb.rq/#b.ib},Rc.wq	Rc←Rav XOR Rbv
ZAP	12.30	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Zap from Rav, byte mask Rbv<7:0>
ZAPNOT	12.31	Ra.rq,{Rb.rq/#b.iq},Rc.wq	Rc←Zap from Rav, byte mask NOT Rbv<7:0>

B

Programming with MACRO–64

This appendix contains information to help you program more effectively using the MACRO–64 assembly language within the framework of the OpenVMS Calling Standard. It discusses the following information:

- Programming hints (Section B.1)
- The OpenVMS Calling Standard (Section B.2)
- Accessing memory with base registers (Section B.3)
- Types of data structures (Section B.4)
- Types of routines (Section B.5)
- Establishing self-addressability (Section B.6)
- Optimization and automatic alignment (Section B.7)

For more information on the AXP environment, you should become familiar with the following manuals:

- *Alpha Architecture Reference Manual*
- *OpenVMS Calling Standard*
- Alpha AXP architecture

B.1 MACRO-64 Programming Hints

To assist you in MACRO–64 programming, several example programs are placed in the SYS$EXAMPLES directory (SYS$EXAMPLES:MACRO64$*.M64) on your system during installation. In order of increasing complexity, these example programs are as follows:

- MACRO64$HELLO.M64—A simple "Hello World" program that demonstrates the calling standard macros.
- MACRO64$WHAMI.M64—A program that displays the WHAMI internal processor register and demonstrates making system service calls from MACRO–64.
- MACRO64$PI.M64—A program that computes the mathematical constant PI to a specified number of digits, and demonstrates general MACRO–64 programming concepts, optimization techniques, and user-defined register symbols.

The remainder of this section contains some hints for programming in MACRO–64.

B.1.1 STARLET.MLB Migration Macros

A large number of macros in STARLET.MLB work correctly with MACRO–64. Some of the macros in STARLET.MLB, however, generate instruction statements designed exclusively for use with the macro migration compiler, the MACRO-32 Compiler for OpenVMS AXP. Because these macros generate VAX instructions, they are not suitable for use with MACRO–64. If you attempt to use one of these VAX instruction-generating macros with MACRO–64, the assembler displays an error message similar to the following:

```
Macro XXXXX is not supported for use with MACRO-64
```

MACRO–64 defines corresponding macros by the same names in MACRO64.MLB, which prevents a series of potentially misleading diagnostics. Because MACRO–64 searches MACRO64.MLB before STARLET.MLB, the macro definitions in MACRO64.MLB occlude those of the same names in STARLET.MLB.

You can still perform the operations with MACRO–64 that these STARLET macros perform. For example, the following STARLET macro invocation does not work with MACRO–64:

```
$ALLOC_S DEVNAM=OUTPUT_DEV_DESCR
```

However, you can use MACRO–64's $CALL macro instead as follows:

```
$CALL SYS$ALLOC, ARGS=<OUTPUT_DEV_DESCR/A>
```

As with other languages, you can call system services from MACRO–64. See Chapter 6 for information on the $CALL macro. Refer to the *OpenVMS System Services Reference Manual* for information on calling OpenVMS system services. After installing MACRO–64, refer to SYS$EXAMPLES:MACRO64$WHAMI.M64 for an example of how to call system services from MACRO–64.

B.1.2 Integer Division

While the Alpha AXP architecture does not provide an instruction for integer division, you can use the OTS$DIV_L and OTS$DIV_I routines to perform quadword and longword integer division, respectively. For an example of how to use these routines, see the MACRO64$PI.M64 example program in the SYS$EXAMPLES directory.

B.1.3 User-Defined Register Symbols and Macros

The macros described in Chapter 6 accept user-defined register symbols in place of the predefined register symbols. When writing your own macros, you may wish to allow for user-defined register symbols as well. To help you do so, the %TYPE( ) lexical operator returns MACRO64$TYPE_GENREG and MACRO64$TYPE_FLTREG for user-defined integer and floating-point registers, respectively. In addition, MACRO-64 supplies you with two lexical operators: %IREG( ) and %FREG( ).

B.2 The OpenVMS Calling Standard

All OpenVMS languages supplied by Digital support the OpenVMS Calling Standard. With most compiled AXP languages, the OpenVMS Calling Standard is invisible to you. However, when programming with the MACRO–64 assembly language, you are responsible for defining the appropriate data structures and using appropriate instruction sequences in order to implement routines that comply with the OpenVMS Calling Standard.

See Chapter 6 for a description of several library macros that are supplied with the MACRO–64 Assembler.

An important role of the OpenVMS Calling Standard is support for debugging, exception handling, and traceback routines. These routines are aided by the OpenVMS Calling Standard data structures and conventions to determine the current routine and the source of its call. In addition, these structures and conventions enable the debugger to interpret other information, such as data stored on the stack or in registers.

Again, when programming in MACRO–64 assembly language, it is important that you define the data structures and adhere to the conventions defined by the OpenVMS Calling Standard.

Example B–1 shows a simple routine, ROUTINE1, that does the following:

- Takes a single, longword parameter as input.
- Calls another function, ROUTINE2, passing its own parameter as a parameter to ROUTINE2.
- Adds the return value from ROUTINE2 and the value of an external longword, I, storing the result in a register.
- Returns the result of the addition to its caller in R0.

Example B–1 Routine Call Example

```
$ROUTINE ROUTINE1, KIND=STACK, SAVED_REGS=<R2,FP>, ARGLIST=<I32>
$LINKAGE_SECTION
I_ADDR: .address I              ; Linkage to external I
$CODE_SECTION
                                ; The prologue instruction sequence
                                ; is generated by $ROUTINE by
                                ; default

; Routine body
.BASE   R27,$LS                 ; Inform assembler that
                                ; R27 -> linkage section
LDQ     R0, I_ADDR              ; R0 -> I
LDL     R2, (R0)                ; R2 <- I
MOV     1, R25                  ; AI <- 1 (R16 already contains
                                ;  the argument)
$CALL   ROUTINE2                ; Call ROUTINE2, R27->linkage sect
ADDL    R2, R0, R0              ; retval <- retval + I
$RETURN                         ; $RETURN generates the epilogue
                                ; sequence.
$END_ROUTINE ROUTINE1
```

B.2.1 Effects on Assembly-Time when Using Calling-Standard Macros

The calling-standard macros provided with MACRO–64 in MACRO64.MLB are designed to make programming in conformance with the OpenVMS Calling Standard significantly easier. See Chapter 6 for a description of these macros.

In general, these macros generate a minimum of instruction and data-directive statements, the job these macros perform is complex and can involve several thousand lines of macro expansion assembly-time statements. When you use these macros you may notice increased assembly time and increased memory

usage at assembly time. Nonetheless, use of the calling-standard macros should significantly speed your program development effort.

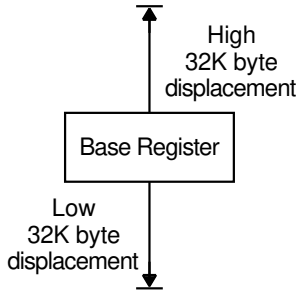
B.3 Accessing Memory with Base Registers

The Alpha AXP architecture is a base-register architecture. Memory locations can only be accessed relative to a base address stored in one of the base registers.

The Alpha AXP architecture has 31 general base registers, R0 through R30. R31 is a special base register, which always reads as 0. Any attempt to store a value in R31 is ignored.

Since memory-reference instructions use a 16-bit, signed displacement relative to the base address, elevated-mode programs may access the low and high 32K bytes of the virtual address space relative to R31. See Figure B–1.

Figure B–1 Base-Register Memory Displacement



NU–2583A–RA

However, these areas of virtual memory are typically not accessible to user-mode programs.

The following sections describe several methods you can use to access memory using base addresses.

Base-Register Architecture

Base-register architectures use different methods to obtain an initial base address for a routine. Typically, base-register architectures load the current program counter (PC) into a base register. Thus, data can be accessed relative to a location in the code. This method is inconsistent with the high-performance objectives of the Alpha AXP architecture for the following reasons:

- It requires placing data areas adjacent to code areas, which results in inefficient use of the instruction and data caches.
- The memory locations near the boundary of the data area and code would likely be duplicated in both the instruction cache and the data cache, which diminishes the effectiveness of the caching algorithms.

Procedure Value

The OpenVMS Calling Standard simplifies obtaining a base address without requiring the data and code areas to be adjacent. When a program or routine needs to call another routine, it must load R27 with a procedure value. The procedure value is a special base address that is the address of the procedure descriptor of the routine it is calling.

Procedure Descriptor

The procedure value for a given routine is the address of the procedure descriptor for that routine. Section B.4.1 describes the procedure descriptor in more detail.

The procedure descriptor is a data block that resides in a special data area owned by the routine. This data area is called the linkage section.

Unlike VAX external routine names, which reference its entry point, AXP external routine names reference the procedure descriptor that contains the entry point's address.

Since the called routine has the address of its own procedure descriptor in R27, it can access not only its own procedure descriptor, but more importantly, it can use that address as a base address and access any data that is within 32K bytes on either side of its procedure descriptor.

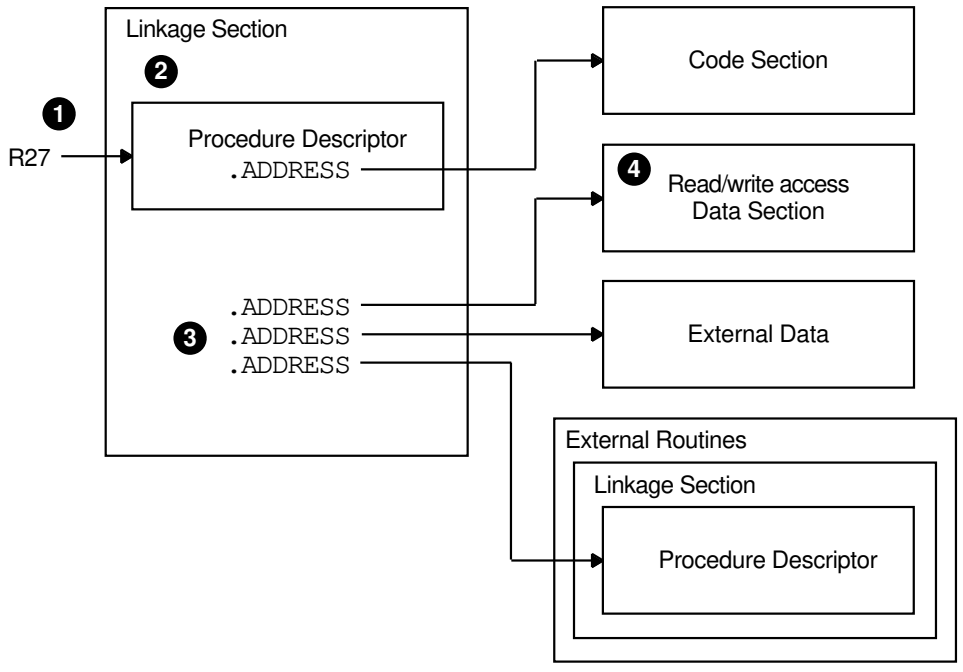
Linkage Section

The special data area where the procedure descriptor resides is called the linkage section. The linkage section is a psect that contains the routine's procedure descriptor. The linkage section performs the following functions:

- Allows other routines to link to the routine using the procedure descriptor.
- Uses it to link to other areas of memory in order to receive and store data.
- Stores the addresses of other routines, allowing the routine to call those routines.

Figure B–2 shows how you access memory using base addresses:

Figure B–2 Accessing Memory Through Base Addresses



❶ The called routine can access memory in its own linkage section relative to R27 since R27 contains a base address within its linkage section, specifically, the address of its procedure descriptor.

❷ Since memory-reference instructions use a 16-bit displacement relative to the base address, the linkage section can be up to 64K bytes in size with direct access relative to R27.

In addition, the linkage section psect typically has the NOWRT attribute and contains only address constants and possibly other constants in order to facilitate placing the linkage section in a shared section in a shareable image.

❸ If the called routine needs to access memory beyond its own linkage section, it can store additional base addresses in its linkage section, load them into base registers as needed, and access any legitimate location in memory.

❹ To access read/write data, a separate read/write data section is defined in a different psect with the WRT attribute and the address of the read/write section is stored in the linkage section.

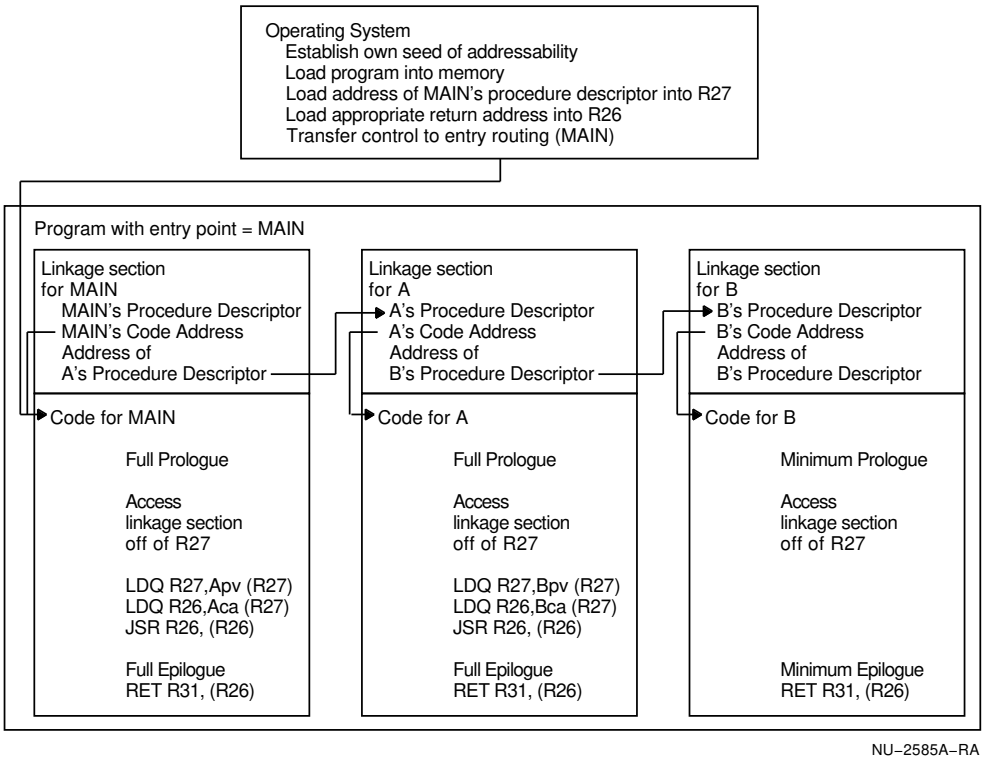
As shown in Figure B–2, the special base address passed in R27 provides the called routine with a seed of addressability from which it can directly access its own linkage section and from which it can indirectly access all of memory. Each routine in a call chain receives its seed of addressability from its caller.

MAIN Program Call Chain

The main routine in a program works in the same way. The operating system passes it a base address within its own linkage section in R27 just like any other calling standard-conforming routine. The operating system can establish its own seed of addressability in a variety of ways. The important programming consideration is that the main routine is given its requisite seed of addressability through the base address passed in R27 when the operating system calls the main routine. The main routine must in turn pass a seed of addressability in R27 to any routines that it calls, and so on down the call chain.

Figure B–3 shows program startup and the program call chain.

Figure B–3 Program Call Chain



NU–2585A–RA

The call chain in Figure B–3 shows a typical program's call chain originating with the transfer to the program's entry point (MAIN) from the operating system. The MAIN routine calls routine A, which in turn calls B. A full prologue and epilogue are required for routines MAIN and A since they both make standard calls. The full prologue allocates the fixed stack area and saves the return address (R26), procedure value (R27), frame pointer (R29), and any preserved registers that are to be used by the routine. Finally, the full prologue establishes the routine's frame as the current frame. If MAIN or A need to access their respective linkage sections after making a standard call, they will save a preserved register in the prologue, move R27 to that register before using R27 in a standard call, use that register to access the linkage section after the call, and restore that register in the epilogue. Since B does not make any standard calls, its prologue and epilogue can be significantly streamlined.

Note that all three routines access data in their respective linkage sections relative to the base address that is in R27 on entry. The caller refers to the callee by the address of the callee's procedure descriptor. The procedure-descriptor address is also called the procedure value (PV), and it is this address that is passed to the callee in R27. To make a call, the caller must store the address of the callee's procedure descriptor in its own linkage section. The caller can then load R27 from its own linkage section before the call. The caller then goes on to obtain the address of the callee's code from the callee's procedure descriptor. With the callee's code address (CA) in R26 and the callee's procedure-descriptor address in R27, the caller can transfer control to the callee with a jump-to-subroutine (JSR) instruction through R26, storing the return address in R26. A more optimal call sequence using a linkage pair is shown in Section B.4.3.

B.4 Types of Data Structures

This section describes the OpenVMS Calling Standard data structures and how to use them in practical applications.

The three data structures defined by the OpenVMS Calling Standard that you should be familiar with are as follows:

- Procedure descriptor
- Signature block
- Linkage pair

B.4.1 Procedure Descriptor

The procedure descriptor contains information that describes the routine. More importantly, it contains the address of the code for the routine. It includes the following information:

- Stack usage (if any)
- Register save/restore masks
- Exception handlers

Usually, a routine's name references the routine's procedure descriptor rather than the address of its code. For example, a global name for an external routine represents the address where the procedure descriptor resides, not the address of the routine's code. Programs that call the routine obtain the address of its code from its procedure descriptor or using a linkage pair.

The .PROCEDURE_DESCRIPTOR and .LOCAL_PROCEDURE_DESCRIPTOR assembler directives define the name of the procedure descriptor, and indicate that the block of storage that follows is the procedure descriptor for the specified routine. .PROCEDURE_DESCRIPTOR and .LOCAL_PROCEDURE_DESCRIPTOR also associate the code address of the routine's entry point with the procedure name. The association between the code address and the procedure name allows the assembler and linker to process the .CODE_ADDRESS and .LINKAGE_PAIR directives. For more information on linkage pairs, see Section B.4.3.

You need to declare the correct amount of storage, and specify the correct initial values. The .PROCEDURE_DESCRIPTOR and .LOCAL_PROCEDURE_DESCRIPTOR directives do not create the storage for the procedure descriptor. Instead, they mark the storage that follows as a procedure descriptor, which results in a special object-module record. Use the $ROUTINE and $PROCEDURE_DESCRIPTOR macros to both mark the procedure descriptor and define the storage for the procedure descriptor. For information on these macros, see Chapter 6. For information on the procedure descriptor format, see the *OpenVMS Calling Standard*.

B.4.2 Signature Block

The signature block describes the parameters, parameter-passing methods, and return value used by the routine. Using the signature block is optional. If it is used, the procedure descriptor references it. You can use the $ROUTINE and $PROCEDURE_DESCRIPTOR macros to define the signature block. For information on these macros, see Chapter 6.

B.4.3 Linkage Pair

You can make calls to routines more efficient by using the .LINKAGE_PAIR and .LOCAL_LINKAGE directives. .LINKAGE_PAIR and .LOCAL_LINKAGE store both the address of the specified routine's code and procedure descriptor (a total of two quadwords).

For example, the assembler provides the .CODE_ADDRESS and .LOCAL_CODE_ADDRESS directives to obtain the address of a routine's code (a quadword). To obtain the address of the routine's code and procedure descriptor, you can use either the .CODE_ADDRESS or .LOCAL_CODE_ADDRESS in conjunction with the .ADDRESS directive. The following example shows how to use these directives to obtain the code address and procedure descriptor of routine X:

```
    .CODE_ADDRESS X         ; Store address of X's code (entry point)
    .ADDRESS      X         ; Store address of X's procedure descriptor
```

.LINKAGE_PAIR performs the same function as using .CODE_ADDRESS and .ADDRESS together, as shown in the following example:

```
    .LINKAGE_PAIR X
```

Using Linkage Pairs in Routines

Another example of how you can use linkage pairs is shown in Example B–2 and Example B–3. Example B–2 shows a code sequence that does not use a linkage pair.

Example B–2 Routine Without Linkage Pairs

```
        .enable local_block

        .psect $LINKAGE,noexe
LS:     .procedure_descriptor MY_PROC, MY_CODE
        ; MY_PROC procedure descriptor details...
X_ADDR: .address X                      ; Store address of X's proc desc

        .psect $CODE,exe
MY_CODE: ; R27 -> LS upon entry
        ; Prologue omitted for clarity...
        LDQ     R27,X_ADDR-LS(R27)      ; R27 -> proc desc for X
        LDQ     R26,8(R27)              ; R26 -> code for X  (1)
        JSR     R26,(R26)               ; CALL X             (2)
        ; Epilogue omitted for clarity...
```

While the previous code sequence works correctly, it has two immediate stalls in the instruction sequence:

(1) The first stall occurs when the second LDQ instruction must wait for the preceding load to R27 to complete.

(2) The second stall occurs when the JSR instruction must wait for the preceding load to R26 to complete.

Since routine calls occur frequently, these stalls significantly increase the amount of time for the routine to execute.

Example B–3 is similar to Example B–2 except it uses a linkage pair to call routine X.

Example B–3 Routine With Linkage Pairs

```
        .enable local_block

        .psect $LINKAGE,noexe
LS:     .procedure_descriptor MY_PROC, MY_CODE
        ;MYPROC procedure descriptor details...
X_LP:   .LINKAGE_PAIR X                 ; Store address of X's code
                                        ; followed by address of X's
                                        ; proc desc

        .psect $CODE,exe
MY_CODE: ; R27 -> LS upon entry
        ; Prologue omitted for clarity...
        LDQ     R26,<X_LP+0>-LS(R27)    ; R26 -> code for X
        LDQ     R27,<X_LP+8>-LS(R27)    ; R27 -> proc desc for X
        JSR     R26,(R26)               ; CALL X
        ; Epilogue omitted for clarity...
```

The same number of instructions are required, but the two immediate stalls in the instruction sequence are eliminated.

You can also use the $LINKAGE_PAIR macro or the $CALL macro. For more information on these macros, see Chapter 6. Note that the $CALL macro generates an instruction sequence similar to that shown in Example B–3.

B.5 Types of Routines

The OpenVMS Calling Standard defines four different types of routines with four different kinds of procedure descriptors.

The format varies slightly for each kind of procedure descriptor. The four types of routines are:

- Null frame
- Register frame
- Stack frame (fixed and variable)
- Bound frame

A **null-frame** routine is a light-weight routine similar in concept to a JSB-linkage routine on a VAX system. A null-frame routine executes in the context of its caller and is therefore restricted in the operations that it can perform. As shown in Table B–1, a null-frame routine requires the least overhead and offers the least capabilities.

A **register-frame** routine creates its own frame and therefore executes in its own context. A register-frame routine is an intermediate-weight routine with intermediate capabilities and overhead. A register-frame routine maintains the context of its callers in registers.

A **stack-frame** routine has two categories: fixed and variable:

- With the fixed-frame stack, the stack-frame size does not vary. A fixed-stack frame routine creates its own frame and stores the context of its caller on the stack. A fixed-stack frame routine is also an intermediate-weight routine with intermediate capabilities and overhead. The only significant capability lacking with a fixed-stack frame routine is the ability to make standard calls.

- With the variable frame stack, the stack-frame size may vary. A variable-stack frame routine is the full function, heavyweight-type of routine. It creates its own stack frame and stores the context of its caller on the stack. The variable-stack frame routine is the only type of routine that can make standard calls.

A **bound-frame** routine is generally used in compiled languages such as Ada and Pascal that require a dynamic, uplevel link to data in an outer routine. Most assembly-language programming does not involve bound-frame routines.

B.5.1 Routine Capabilities

The types of operations that a routine may perform and the amount of required overhead in the routine's prologue and epilogue instruction sequences are different for each type of routine.

The capabilities and overhead requirements of null-frame, register-frame, fixed-stack frame, and variable-stack frame routines are summarized in Table B–1.

Table B–1 Frame Attributes

	Null Frame	Fixed-Register Frame	Variable-Register Frame	Fixed-Stack Frame	Variable-Stack Frame
Executes in Frame of Caller	Yes	No	No	No	No
Where Caller's Context is Kept	–	Registers	Registers	Stack	Stack
Can Allocate Stack in Proloque	–	Yes	Yes	Yes	Yes
Must Allocate Stack in Prologue	–	No	No	Yes	Yes
Can Allocate Stack in Routine Body	Restricted	No	Yes	No	Yes
Can Have an Exception Handler	No	Yes	Yes	Yes	Yes
Can Save/Restore Registers	Restricted	No	No	Yes	Yes
Can Make Standard Calls	Restricted	No	No	No	Yes

For more information on routines, see the *OpenVMS Calling Standard*.

B.5.2 Entry Prologue and Exit Epilogue Sequences

In general, a standard sequence of instructions is used when a routine first starts and when it completes. The following code example shows a typical entry prologue sequence for a variable stack-frame procedure. Several elements of the prologue may be omitted for less complex routines.

```
ENTRY:  LDA     SP, -32(SP)     ; Allocate fixed-stack area  ❶
        STQ     R27, (SP)       ; Save procedure value
        STQ     R26, 8(SP)      ; Save return address
        STQ     R2, 16(SP)      ; Save any preserved regs that are used
        STQ     FP, 24(SP)      ; Save old FP
        MOV     SP, FP          ; Establish current frame
        $END_PROLOGUE           ; Marks the end of the proloque ❷
```

❶ The ENTRY label marks the code entry point for the routine.

❷ The $END_PROLOGUE macro is used to mark the end of the prologue.

The following code example shows a typical exit epilogue sequence for a variable-stack frame procedure. Several elements of the epilogue may be omitted for routines of less complexity.

```
$BEGIN_EPILOGUE         ; Mark the beginning of the epilogue ❶
MOV     FP, SP          ; Release the variable stack area
LDQ     R28, 8(FP)      ; Fetch return address
LDQ     R2, 16(FP)      ; Restore preserved registers
LDQ     FP, 24(FP)      ; Reestablish caller's frame
LDA     SP, 32(SP)      ; Release the fixed-stack area
RET     R28             ; Return to caller
$END_EPILOGUE           ; Mark the end of the epilogue ❶
```

❶ In this example, the $BEGIN_EPILOGUE and $END_EPILOGUE macros are used to begin and end the epilogue sequence.

The $ROUTINE macro optionally generates a standard prologue instruction sequence at the beginning of the routine. In addition, you can also use the $RETURN macro to generate a standard epilogue sequence. For more information on these macros, see Chapter 6.

B.6 Establishing Self-Addressability

A MACRO–64 routine can establish addressability to its own linkage section without the help of its caller (the calling routine). In this case, the caller does not pass the procedure value in R27 as in a standard call. Therefore, the routine must establish its own base address within its linkage section.

To accomplish self-addressability, the code and linkage sections must be adjacent to each other. This requires them to reside in the same psect. Because the assembler does not allow placement of data in an EXE/NOMIX psect, instructions must be placed in an EXE/MIX psect. When debugging, the debugger views execution to have transferred into a data area because the debugger views all MIX psects as data.

NOTE

The debugger can correctly display decoded instructions in a MIX psect, but it cannot achieve source-line /program-counter correlation and source-line scrolling.

Example B–4 represents the CLEAR_X_ARRAY routine, which has no procedure descriptor.

Example B–4 Establishing a Base Address

```
        .psect CLEAR_X_ARRAY_CODE,exe,mix,quad
X_ARRAY_ADDR:
        .address X_ARRAY
X_ARRAY_SIZE:
        .long <X_ARRAY_END - X_ARRAY> / 8
```

(continued on next page)

Example B–4 (Cont.) Establishing a Base Address

```
CLEAR_X_ARRAY::
        BSR     R1,10$                  ; R1 -> 10$
10$:    LDQ     R22,X_ARRAY_ADDR-10$(R1); R22 -> X_ARRAY
        LDL     R23,X_ARRAY_SIZE-10$(R1); R23 <- REMAINING
20$:    CLR     R31,(R22)               ; [X_ARRAY_PTR] <- 0
        LDA     R22,8(R22)              ; X_ARRAY_PTR<-X_ARRAY_PTR+8
        LDA     R23,-1(R23)             ; REMAINING<-REMAINING-1
        BGT     R23,20$                 ; Branch if REMAINING > 0
        RET     (R26)                   ; Return
        .end
```

Other routines may call the CLEAR_X_ARRAY routine using only its code address and a nonstandard linkage. The CLEAR_X_ARRAY routine executes in the context of its caller; it does not create its own frame. It establishes a base address in its code section and because its linkage section is adjacent to its code section, it can access data in its linkage section.

B.7 Optimization and Automatic Alignment

MACRO–64 provides the following optimizations:

- Automatic data alignment
- Automatic code alignment
- Scheduling
- Peepholing

By default, all optimizations and automatic alignment are off. When invoking MACRO–64, the default qualifiers are /NOOPTIMIZE and /NOALIGNMENT.

You can control optimization and automatic alignment using the following qualfiers and directives:

- The /OPTIMIZE and /ALIGNMENT qualifiers
- The .ENABLE and .DISABLE directives
- The .BEGIN_EXACT and .END_EXACT directives

The .ENABLE and .DISABLE directives control optimization and automatic code alignment for an assembly unit; you cannot use these directives to control optimization and automatic code alignment for a range of statements. Use .BEGIN_EXACT and .END_EXACT to suppress optimization for a range of statements.

B.7.1 Automatic Data Alignment

Access to naturally aligned data is faster than access to unaligned data. MACRO–64 does not align data by default. Instead, it places data exactly as specified with no padding. With automatic data alignment selected, data directives are automatically aligned on a natural boundary. Pad bytes are added as necessary to achieve natural alignment. Labels that occur with the data are automatically aligned with the data.

B.7.1.1 Controlling Data Alignment

You can control when you want to have data alignment in your program. You do this by using the .ENABLE ALIGN_DATA and .DISABLE ALIGN_DATA directives. Note that automatic data alignment is disabled by default. Example B–5 shows how to enable and disable data alignment.

Example B–5 Enabling and Disabling Data Alignment

```
.PSECT D1,DATA,NONEXE
.DISABLE ALIGN_DATA     ; This is the default state ❶
.BYTE   1               ; offset = 0
.QUAD   2               ; offset = 1
.PSECT D2,DATA,NOEXE
.ENABLE ALIGN_DATA      ; Turn on automatic alignment ❷
.BYTE   1               ; offset = 0
.QUAD   2               ; offset = 8 (pad bytes are added at offsets 1 - 7)
```

❶ The .DISABLE ALIGN_DATA directive causes the assembler to allocate data at the current location without padding, regardless of whether it is within a natural boundary.

❷ The .ENABLE ALIGN_DATA directive causes the assembler to align data objects on natural rather than byte boundaries.

B.7.1.2 Directives for Automatic Data Alignment

The MACRO–64 data-storage directives in Table B–2 provide optional, automatic alignment.

Table B–2 Directives Using Automatic Alignment

Directive	Alignment Boundary
.WORD	word (2)
.SIGNED_WORD	word (2)
.LONG	longword (4)
.ADDRESS	quadword (8)
.OCTA	octaword (16)
.QUAD	quadword (8)
.CODE_ADDRESS	quadword (8)
.LOCAL_CODE_ADDRESS	quadword (8)
.LINKAGE_PAIR	octaword (16)
.LOCAL_LINKAGE_PAIR	octaword (16)
.PROCEDURE_DESCRIPTOR	quadword (8)
.LOCAL_PROCEDURE_ DESCRIPTOR	quadword (8)
.F_FLOATING (.FLOAT)	longword (4)
.D_FLOATING (.DOUBLE)	quadword (8)
.G_FLOATING	quadword (8)
.S_FLOATING	longword (4)

(continued on next page)

Table B–2 (Cont.) Directives Using Automatic Alignment

Directive	Alignment Boundary
.T_FLOATING	quadword (8)
.BLKA	quadword (8)
.BLKD	quadword (8)
.BLKF	longword (4)
.BLKG	quadword (8)
.BLKL	longword (4)
.BLKO	octaword (16)
.BLKQ	quadword (8)
.BLKS	longword (4)
.BLKT	quadword (8)
.BLKW	word (2)
.ASCID	quadword (8)
.INSTRUCTION	longword (4)

Specify a .PSECT directive alignment attribute that aligns the psect on a boundary that is at least as stringent as the automatically aligned data items.

B.7.2 Automatic Code Label Alignment

Automatic code label alignment improves I-cache utilization and multi-instruction issue. This optimization aligns certain code labels to a quadword boundary by padding with NOP and FNOP instructions. Use the /ENVIRONMENT=NOFLOAT qualifier to suppress FNOP instructions.

Code labels that are the target of backward branches are aligned. Also, labels that are the target of forward branches that are not also the target of a fall through are aligned.

Use the /ALIGNMENT=CODE qualifier or the .ENABLE ALIGN_CODE directive to select automatic code label alignment.

B.7.3 Scheduling Optimization

Scheduling reorders instructions to reduce pipeline stalls and increases the chances for multi-instruction issue. It only reorders instructions; it does not change the instructions or the registers used in the instructions. In addition, instructions are not scheduled across labels. Therefore, the structure of your program is retained.

Note

If you use the same register for two or more logically distinct actions, the scheduler schedules these actions serially in the sequence you specify. If you use different registers for two or more logically distinct actions, the scheduler schedules the actions in parallel, which provides faster execution performance.

Use the /OPTIMIZE=SCHEDULE qualifier or the .ENABLE SCHEDULE directive to select scheduling optimization.

B.7.4 Peephole Optimization

Peephole optimization restructures your program for faster execution. To do so, it reduces the number of instructions executed and replaces long-executing instructions with instructions that require less execution time. Although this optimization may increase the number of instructions in memory, the number actually executed, as you follow the code paths, decreases or remains the same. Peephole optimization matches short code patterns and replaces them with more optimal sequences. It also eliminates dead code. (Note that dead code is often created by the code changes made by peephole optimization. This dead code is eliminated by successive iterations where peephole optimization is selected.) Peephole optimization eliminates some branches by inlining or replicating short sequences from the branch target.

Use the /OPTIMIZE=PEEPHOLE qualifier or the .ENABLE PEEPHOLE directive to select peephole optimization.

B.7.5 Using MACRO–64 for Performance

High-order language compilers do optimizations called loop unrolling and inlining. MACRO–64 allows you to perform these optimizations manually through coding techniques, although it does not offer specific qualifiers or directives for these optimizations.

Loop unrolling reduces the number of branches that must be executed. You can unroll loops using the .REPEAT directive. The following example shows manual loop unrolling:

```
10$:                          ; Copy loop
.repeat 10                    ; Unroll the loop, factor=10
    LDA     R18, -1(R18)      ; Decrement quadword count
    BLT     R18, 20$          ; Branch if quadword count < 0
    LDQ     R0, (R17)         ; Load a quadword
    STQ     R0, (R16)         ; Store the quadword
    LDA     R16, 8(R16)       ; Increment the target address
    LDA     R17, 8(R17)       ; Increment the source address
    BR 10$
.endr
20$:
```

Loop unrolling complemented with MACRO–64 scheduling and peephole optimizations can significantly increase performance. For better loop scheduling, you can make the counter decrement and the loop exit separable.

```
10$:                          ; Copy loop
.repeat 10                    ; Unroll the loop, factor=10
    LDA     R18, -1(R18)      ; Decrement quadword count
    LDQ     R0, (R17)         ; Load a quadword
    BLT     R18, 20$          ; Branch if quadword count < 0
    STQ     R0, (R16)         ; Store the quadword
    LDA     R16, 8(R16)       ; Increment the target address
    LDA     R17, 8(R17)       ; Increment the source address
    BR 10$
.endr
20$:
```

High-order language compilers often inline a routine call if the called routine is small and is defined in the same module from which it is called. To inline a routine call, the compiler replaces the routine call with the body of the called routine. You can use macros to manually inline short, local routines with MACRO–64.

B.7.6 Viewing Optimization Results

You can view the results of optimization by comparing code from before and after optimization. Use the /LIST and /MACHINE_CODE command-line qualifiers described in Section 1.2.1. Using these qualifiers provides you with a binary machine-code listing, which you can also use to view the results of automatic alignment features.

Many of the other OpenVMS AXP languages offer a /MACHINE_CODE command-line qualifier that shows the assembly language generated for your source program in the listing file. You can often use this feature to learn about AXP assembly language and optimization techniques.

C
Using LSE with MACRO–64

This appendix explains how to use the DEC Language-Sensitive Editor (LSE) with the MACRO–64 language. For LSE to function correctly, LSE must be installed before the MACRO–64 assembler.

C.1 Invoking LSE

To use LSE with the MACRO–64 language, you must name your program source file with a .M64 file extension. For example, to edit a file named TEST.M64, enter the following command:

```
$ LSE TEST.M64
```

Using the .M64 file extension invokes the LSE editor and places you in the MACRO–64 language environment.

If you choose to use a different file extension, you can still use LSE in the MACRO–64 language environment. For example, to edit a file named TEST.ASM, enter the following command:

```
$ LSE/LANGUAGE=MACRO64 TEST.ASM
```

If you use LSE with MACRO–64, and you want the same behavior on the OpenVMS AXP operating system as on the OpenVMS VAX operating system, enter the following LSE command:

```
$ SET COMMAND LANGUAGE VMS
```

C.2 Running Diagnostics

You can run diagnostics in LSE to debug programs without leaving the LSE editor. For more information, see the *Guide to Language-Sensitive Editor for VMS Systems* and the MACRO–64 release notes.

When running diagnostics in an LSE editing session, MACRO–64 displays error messages of different severity levels. For more information on error messages, see Appendix E.

D
Differences from VAX MACRO

MACRO–64 is a new product that provides a macro assembly language for the native Alpha AXP instruction set. The MACRO–64 instruction set is completely different from the VAX MACRO instruction set. Although many of the directives are the same between VAX MACRO and MACRO–64, it would not be practical to share instruction sequences between the two assembly languages.

While MACRO–64 is compatible with VAX MACRO, even allowing shared source modules between VAX MACRO and MACRO–64 using common assembler directives and through conditional assembly, several differences exist. This appendix contains two sections describing the differences in assembly directives and features as follows:

- Section D.1 describes new features in MACRO–64 that are not present in VAX MACRO.
- Section D.2 describes features in VAX MACRO that are not present in MACRO–64.

D.1 Assembler Features in MACRO–64

This section describes MACRO–64 features not present in VAX MACRO. They are:

- .BASE directive

  The .BASE directive is provided for implicit base register support.

- .PROCEDURE_DESCRIPTOR and .LOCAL_PROCEDURE_DESCRIPTOR directives

  These directives mark the specified identifier as a procedure descriptor. .PROCEDURE_DESCRIPTOR declares an identifier as a global label that defines the address of the procedure's description (a data address) and associates the procedure's entry point (a code address) with the name of the procedure. .LOCAL_PROCEDURE_DESCRIPTOR performs a similar function by defining a local label rather than a global label. No storage is allocated. For more information on these directives, see Chapter 5.

- .LINKAGE_PAIR and .LOCAL_LINKAGE_PAIR directives

  These directives cause a linkage pair to be stored at the current psect and current location counter. A linkage pair consists of a code address and the address of the specified identifier. For more information on these directives, see Chapter 5.

- .CODE_ADDRESS and .LOCAL_CODE_ADDRESS directives

  These directives cause the code address(es) of the specified identifier(s) to be placed at the current psect and current location counter. See Chapter 5 for more information.

- .ELSE directive

  MACRO–64 introduces a new conditional-assembly directive, .ELSE. You can use it in conjunction with the .IF and .ENDC directives (common to VAX MACRO and MACRO–64) to specify a sequence of statements to be assembled when the condition in the controlling .IF directive is false. While the .ELSE directive is similar to VAX MACRO's .IF_FALSE directive, it differs in that you may specify at most one .ELSE directive within an .IF /.ENDC block. .IF_FALSE is also provided for VAX MACRO compatibility. See Chapter 5 for more information.

- .INCLUDE directive

  MACRO–64 provides a new .INCLUDE directive that causes a specified file to be textually included in the assembly. The format of this directive is as follows:

  ```
  .INCLUDE "file_specification" ["Directory_search_list"]
  ```

  The effect of the .INCLUDE directive is similar to that of the #include directive in C or the REQUIRE statement in BLISS.

- Optional case sensitivity in identifiers

  You can specify the /NAMES=AS_IS command-line qualifier to cause the assembler to distinguish between uppercase and lowercase letters in identifiers and external symbol names. The default is /NAMES=UPPERCASE; uppercase and lowercase letters are not distinguished as with VAX MACRO. See Chapter 1 for more information.

- Lexical operator support for 12 lexical operators

  MACRO–64 provides 12 lexical operators, three of which are compatible with VAX MACRO's three macro string operators. You can use MACRO–64 lexical operators in any context and you are not restricted to macros. See Chapter 3 for more information.

- Lexical string symbols

  See Chapter 3 for more information.

- Lexical escape operator (%%)

  See Chapter 3 for more information.

- Lexical substitution operator (%lexsym_name%)

  See Chapter 3 for more information.

- More liberal use of the value-of operator (\)

  MACRO–64 as well as VAX MACRO recognizes the value-of operator (\) in macro invocations and in default-value strings for parameters in .MACRO directives. In addition, MACRO–64 recognizes the value-of operator (\) in the following contexts:

  - With either or both of the two arguments in the .IF DIFFERENT and .IF IDENTICAL directives.
  - With the argument to the .IF BLANK and .IF NOT_BLANK directives.
  - With the second argument to the .NCHR and .IRPC directives.
  - With the second and subsequent arguments to the .IRP directive.
  - With arguments to lexical operators. See Chapter 4 for more information.

- MACRO64$ symbol

  MACRO–64 implicitly defines a new symbol, MACRO64$, for use in conditional assembly. This symbol is defined in all uppercase and all lowercase letters when /NAMES=AS_IS is specified on the command line. This symbol might be used as follows:

```
.MACRO ADDRESS32 EXPR_LIST
    .IF DEFINED MACRO64$
    .LONG EXPR_LIST
    .IF_FALSE
    .ADDRESS EXPR_LIST
    .ENDC
.ENDM ADDRESS32
ADDRESS32 <A,B,C>
```

- Automatic data alignment

  .ENABLE_ALIGN_DATA causes data to be naturally aligned. See Chapter 5.

- Optimizations

  The assembler optionally performs a number of optimizations. See Chapter 2 (command line) and Chapter 5 (.ENABLE).

- $OPDEF macro

  $OPDEF is supplied in place of VAX MACRO's .OPDEF directive.

- OpenVMS Calling Standard macros:

$ROUTINE	$SEND_PROLOGUE
$CALL	$LINKAGE_SECTION
$CODE_SECTION	$DATA_SECTION
$LINKAGE_PAIR	$BEGIN_EPILOGUE
$RETURN	$END_EPILOGUE
$PROCEDURE_DESCRIPTOR	$RESET_LP_LIST

- .INSTRUCTION directive

  For more information on the .INSTRUCTION directive, see Chapter 5.

- .BEGIN_EXACT and .END_EXACT directives

  For more information on these directives, see Chapter 5.

- /[NO]DEFINE qualifier

  MACRO–64 includes a new command-line qualifier, /[NO]DEFINE. It defines one or more numeric symbols by performing the same function as direct symbol assignment in your source program. For more information about this qualifier, see Chapter 1.

- Preprocessor output

  MACRO–64 provides a new command-line qualifier, PREPROCESSOR_ONLY and a new symbolic argument for the .ENABLE and .DISABLE directives, PREPROCESSOR_OUTPUT, that you can use to produce a preprocessor-output file.

  For more information about the /PREPROCESSOR_ONLY command-line qualifier, see Chapter 1. For more information about the PREPROCESSOR_ OUTPUT argument, see the descriptions of the .ENABLE and .DISABLE directives in Chapter 5.

- Register symbols

  Unlike VAX MACRO register symbols, MACRO–64 symbols are not permanently reserved. You can delete the definition of any predefined register symbol (using the .UNDEFINE_REG directive) or define your own (using the .DEFINE_FREG or .DEFINE_IREG directive).

  For more information about register symbols, see Chapter 1. For a list of OpenVMS Calling Standard register symbols you can use with MACRO–64 supplied macros, see Chapter 6. For more information about defining register symbols and deleting register symbol definitions, see Chapter 5.

D.2 VAX MACRO Features Not Present in MACRO–64

This section contains features in VAX MACRO that are not present in MACRO–64. They are:

- EXE/NOEXE restriction

  MACRO–64 allows only instructions to be placed in a psect with either the EXE or MIX attributes or both. You can place data in a psect with either the NOEXE or MIX attributes or both. VAX MACRO allows instructions or data to be arbitrarily placed in psects.

- Setting the current location in an EXE/NOMIX psect

  MACRO–64 does not permit the current location counter to be changed in a psect with the EXE and NOMIX attributes. VAX MACRO does not have this restriction.

- Overlapping and replacement initializers

  VAX MACRO allows you to reassign the current location counter and store data initial values and instructions over previous data initial values and instructions. MACRO–64 provides a similar capability. However, this capability is only available in psects that have either the NOEXE or MIX attribute, or both. Furthermore, a replacement initial value must start at exactly the same psect offset and be of exactly the same length as the initial value it replaces. VAX MACRO does not have these restrictions.

- Radix unary operators

  VAX MACRO allows unary radix operators on expressions. MACRO–64 allows unary radix operators only to specify the radix of numeric constants. The octal constant ^O123 is valid with both VAX MACRO and MACRO–64. The octal expression ^O(10 + 10) is valid with VAX MACRO; it is not valid with MACRO–64.

- The register mask operator

  The VAX MACRO ^M operator is not supported on MACRO–64. Because register masks are not needed by any MACRO–64 instructions or directives, there is no need for this operator.

- .ALIGN does not support PAGE

  Because the Alpha architecture has variable page sizes, MACRO–64 does not support PAGE as an argument to the .ALIGN directive.

- No count specifiers on data-storage directives

  MACRO–64 does not allow a repeat count expression on any data-storage directive such as .BYTE or .WORD. You can use the %REPEAT lexical function for this purpose.

- No support for the .CROSS and .NOCROSS directives

  There is no support for these directives in MACRO–64.

- No support for .DEBUG

  There is no support for this directive in MACRO–64. However, even though this directive is not supported, MACRO–64 provides symbolic information in the object module for use by the debugger.

- No .DEFAULT directive

  This directive in not applicable on MACRO–64.

- .ENABLE/.DISABLE directive arguments

  The MACRO–64 assembler does not support the following arguments to the .ENABLE or .DISABLE directives:

  - ABSOLUTE—not applicable on MACRO–64
  - DEBUG
  - SUPPRESSION
  - TRACEBACK
  - TRUNCATION

- .END directive differences

  VAX MACRO requires the .END directive. MACRO–64 does not require this directive. MACRO–64 requires the optional identifier that follows the .END directive to be a previously declared procedure descriptor.

- .ENTRY directive not supported

  A $ROUTINE macro that accomplishes a similar function is used.

- No .GLOBAL directive

  There is no support for this directive. VAX MACRO provided this directive for MACRO–11 compatibility only.

- No .H_FLOATING directive

  Since this data type is not a native MACRO–64 type, this directive is not supported.

- No .LINK directive

  There is no support for this directive.

- No .MASK directive

  This directive is not applicable to MACRO–64.

- No .NTYPE directive

  This directive is not supported by MACRO–64. The %TYPE lexical operator provides a similar function.

- No .OPDEF directive

  There is no support for this directive. The $OPDEF macro provides a similar function.

- Differences in .PSECT directive

  The following .PSECT arguments are not supported: PAGE, LIB, and USR.

- No .REFn directives

There is no support for these directives.

- Differences in .RESTORE directive

  On successive .RESTOREs for the same psect, VAX MACRO restores the current location counter value as the last location counter value saved for the psect being restored. MACRO–64 restores the current location counter value as the location counter value saved for the matching .SAVE directive.

- No .TRANSFER directive

  There is no support for this directive.

- Differences with .ENDM and .ENDR

  VAX MACRO considers the .ENDM and .ENDR directives to be synonyms of each other. MACRO–64 considers .ENDM and .ENDR to be distinct. You must end macro definitions with .ENDM and you must end repeat ranges with .ENDR.

E
Error Messages

This appendix describes the messages produced by the MACRO–64 Assembler for OpenVMS AXP Systems.

The description of each message gives the severity, followed by additional explanatory text and suggested action.

ADDTRUNC, Storing an address expression into a storage allocation less than the size of an address results in data truncation.

Informational: The assembler stored a value that is too large for the allocated space, resulting in data truncation.

User Action: Allocate more storage.

ALIGNFILLIGN, The optional .ALIGN fill pattern argument is ignored in psects with the EXE and NOMIX attributes.

Warning: The optional fill pattern is ignored because it is only valid for psects that do not possess the EXE and NOMIX attributes.

User Action: Omit the fill pattern or specify the MIX psect attribute.

ALIGNFILLTRUNC, The value you specify for the .ALIGN optional fill pattern must be an integer in the range of 0 . . . 255. Data truncation occurs with the currently specified fill pattern in a byte storage location.

Warning: The value you specify as the fill pattern for the .ALIGN directive must be within the range of 0 . . . 255. Data truncation occurs whenever you specify a value that is outside of this range.

User Action: Specify a smaller value for the fill pattern.

ALIGNLABELIGN, The ALIGN_LABEL option has been replaced by the ALIGN_CODE option.

Error: The ALIGN_LABEL option has been replaced by the ALIGN_CODE option.

User Action: Use the recommended new option.

ALIGNTOBIG, Specified alignment is too large for PSECT.

Error: The alignment you specified is too large for the current psect.

User Action: Check the psect attributes to insure that the psect alignment is greater than or equal to the alignment you are requesting.

ASCIITRUNC, ASCII constant contains too many characters; value is truncated.

Error: Your ASCII constant contains more than eight characters with the ^A or ^a radix specifier. The assembler deletes the extra characters.

User Action: Check your source code. Use eight or less characters.

BADALIGN, Alignment specifier is out of range.

Error: The alignment specifier used with the .PSECT or .ALIGN directive is out of range.

User Action: For a description of the .PSECT and .ALIGN directives, see Chapter 5.

BADENDARG, Bad argument to .END directive.

Error: The optional argument to the .END directive is invalid.

User Action: If you specify the argument, it must reference a procedure descriptor within the module. Specify a valid procedure descriptor name or omit the argument.

BADINSARG, Argument N is invalid for this instruction.

Error: The argument number shown is invalid for the instruction.

User Action: Check the argument and required format as specified in the documentation.

BADLIB, Error opening library file XXXXX.

Error: The assembler encountered an error when attempting to open the indicated library file.

User Action: Check the file format and file protections.

BADMACPARAMNAME, Illegal macro parameter name.

Error: The indicated macro parameter name is illegal.

User Action: Examine your source code and see Chapter 1 for information about valid parameter names.

BADMACRONAME, Illegal macro name.

Error: The indicated macro name is illegal.

User Action: Check your source code and see Chapter 2 for information about valid macro names.

BADOPERAND, Invalid operand type for operator.

Error: The resolved operand type is invalid for the specified operator.

User Action: See the descriptions of operators, operands, and expressions in Chapter 2.

BADPARAMSTR, Illegal parameter string.

Error: The string specified as a macro parameter is invalid.

User Action: Examine your source code and see Chapter 1 for information about valid parameter names.

BADSYSCALL, Internal error. Bad system call.

Error: The assembler encountered an unexpected internal error when performing a system call.

User Action: Report the problem to Digital.

BASEFAIL, Argument N is invalid. The assembler failed to find a base register specified with a previous .BASE directive to form a register expression of the form offset(Rn).

Error: The assembler could not find a base register, which you specified with a previous .BASE directive, to form a valid register expression of the form offset(Rn).

User Action: Check the instruction in the source code and see Chapter 5 for information about using the .BASE directive.

BASERANGE, Argument N invalid. The assembler attempted to use base register Rn to form a register expression of the form offset(Rn). However, the argument offset exceeds the allowable range of −32,768 to +32,767.

Error: The assembler attempted to use a base register, which you specified with a previous .BASE directive, to form a valid register expression of the form offset(Rn). This attempt failed because the specified argument offset exceeded the valid range of the base register offset (_32,768 to +32,767). The register cited in the message represents the register that produced an offset closest to the range of −32,768 to +32,767.

User Action: Check the instruction in the source code and see Chapter 5 for information about using the .BASE directive.

BEGEXPSC, .BEGIN_EXACT is invalid in a psect with the NOEXE and NOMIX attributes.

Error: A .BEGIN_EXACT directive is not valid in a psect with the NOEXE and NOMIX attributes.

User Action: Check your source code.

BYTEALIGNIGN, The BYTE_ALIGN option has been replaced by the ALIGN_DATA option.

Error: The BYTE_ALIGN option has been replaced by the ALIGN_DATA option.

User Action: Use the recommended new option.

CONPSECTATTR, Contradictory PSECT attribute.

Error: A previously specified psect attribute conflicts with the flagged psect attribute.

User Action: See Chapter 5 for a description of the .PSECT directive and psect attributes.

CONTEOF, Assembler encountered end of file after line continuation.

Error: The assembler encountered end of file after a line that specified a continuation.

User Action: Check your source code.

DATAALIGNTOBIG, Data requires alignment too large for PSECT.

Error: The alignment required for a specified data item is too large for the psect.

User Action: Check the psect attributes to insure that the psect alignment is greater than or equal to the required alignment of the data items. See Chapter 5 for a description of the .PSECT directive and psect attributes.

DATANOTINNOEXE, Data declarations must be in a psect with the MIX or NOEXE attribute.

Error: A data declaration, such as a data-storage directive, has been specified in a psect with incorrect psect attributes.

User Action: Make sure the psect has the MIX or NOEXE attribute set. See Chapter 5 for a description of the .PSECT directive and psect attributes.

DIRNOTINNOEXE, Directive must be in a psect with the MIX or NOEXE attribute.

Error: The directive you specify must appear in a psect with the MIX or NOEXE attribute set.

User Action: Make sure you specify a psect with the MIX or NOEXE attribute set. See Chapter 5 for a description of the .PSECT directive and psect attributes.

DISPTOOLGE, Branch offset is too large for this instruction.

Error: The offset you specified is too large for this instruction.

User Action: Check the range of the specified target to insure it falls between −1048576 . . . +1048575, inclusive.

DUPLEXTERN, External item has multiple definitions.

Error: The item you declared as externally defined with the .EXTERNAL attribute has another conflicting definition within this assembly unit.

User Action: Check the definitions for the specified item.

DUPLGLOBAL, Duplicate global name.

Warning: The assember detected a duplicate global name.

User Action: Check all references in your source code to this name.

DUPMACPARAMNAME, Duplicate macro parameter name.

Error: The assembler detected a duplicate macro parameter name.

User Action: Check your source code.

ENDEXPSC, .END_EXACT is invalid in a psect with the NOEXE and NOMIX attributes.

Error: A .END_EXACT directive is not valid in a psect with the NOEXE and NOMIX attributes.

User Action: Check your source code.

EOLEXP, Assembler expected an end of line.

Error: The assembler expected no more input from the current line.

User Action: Check your source code.

ESCAPE, Illegal escape sequence in string literal; assembler expected \, ", x, or X.

Error: The escape sequence you specified in the string literal is illegal.

User Action: Check your source code.

EXP32BITTRUNC, Assembler expected an integer in the range 0 . . . (2^32) −1 for an unsigned expression OR −(2^31) . . . +(2^31)−1 for a signed expression. Data truncation to 32 bits.

Warning: The assembler found an integer that was not within the expected range.

User Action: Check your source code. The literal must be within the range of 0 . . . (2^32)−1 for an unsigned expression OR −(2^31) . . . +(2^31)−1 for a signed expression. Data truncation to 32 bits occurs.

EXP32BITTYPE, Assembler expected an integer in the range 0 . . . (2^32)−1 for unsigned expression OR −(2^31) . . . +(2^31)−1 for signed expression.

Error: The assembler expected an unsigned integer value within the range of 0 . . . (2^32)−1 or a signed integer value within in the range of −(2^31) . . . +(2^31)−1.

User Action: Check your source code.

EXPBINEXPTERM, Assembler found XXXXX when expecting a binary operator or expression terminator.

Error: The assembler expected a binary operator, such as the plus sign (+) for binary addition, or an item to end the expression, such as the right-angle bracket (>).

User Action: Check the flagged item in the source statement.

EXPFPREG, Argument N is invalid. Assembler expected a floating point register.

Error: The instruction argument cited is invalid. The assembler expected a floating-point register.

User Action: Check your source code and the instruction documentation.

EXPGENREG, Argument N is invalid. Assembler expected a general register.

Error: The instruction argument cited is invalid. The assembler expected a general register.

User Action: Check your source code and the instruction documentation.

EXPIDPROC, Argument N is invalid. Assembler expected an identifier representing a procedure value.

Error: The argument cited is invalid. The assembler expected a user identifier that represents a procedure value.

User Action: Check your source code and the instruction documentation.

EXPINTPAL, Assembler expected an integer expression or PAL opcode.

Error: Integer expession or PAL opcode missing.

User Action: Replace the flagged item with an integer or PAL opcode.

EXPLAB, Argument N is invalid. Assembler expected a label defined in the same psect.

Error: The cited argument is invalid. The assembler expected a label definition to occur in the same psect as its reference.

User Action: Check your source code and the instruction documentation.

EXPLITVAL, Argument N is invalid. Assembler expected an integer literal value in the inclusive range 0 . . . 255.

Error: The instruction argument cited is invalid. The assembler expected an integer literal within the range of 0 . . . 255.

User Action: Check your source code and the instruction documentation.

EXPMACRONAME, Assembler expected a valid macro name.

Error: The assembler expected a valid macro name in this context.

User Action: Check your source code to insure that the item flagged is a user identifier, opcode, or nonmacro directive.

EXPPALOPLIT, Argument N is invalid. Assembler expected an integer literal value in the inclusive range 0 . . . 67108863.

Error: The instruction argument cited is invalid. The assembler expected an integer literal.

User Action: Check your source code and the instruction documentation.

EXPREGOFF, Argument N is invalid. Assembler expected a general register expression of the form offset(Rn).

Error: The cited argument is invalid. The assembler expected a general register expression of the form integer_offset(Rn) for this argument.

User Action: Check the source code and the instruction documentation.

EXPRESEXP, Argument N is invalid. Assembler expected an expression with no forward references resolvable to psect +/ − offset.

Error: The argument cited is invalid. The assembler expected an expression with no forward references.

User Action: Check your source code and the instruction documentation.

EXPSTACKOVER, Internal SEM expression stack overflow.

Fatal: An internal error has occurred.

User Action: Gather as much information as possible about the circumstances under which the error occurred and report the problem to Digital.

EXPTOOCMPLX, Expression is too complex to evaluate.

Error: The expression is too complex for the assembler to evaluate.

User Action: Try grouping the expression components using angle brackets (< >). The most complex expression form handled by the assembler resolves to the form: <psect/symbol +/− offset> OPERATOR <psect/symbol +/− offset>, where OPERATOR is one of: +, −, *, /, @, \, &, or !. See Chapter 2 for further descriptions of the assembler evaluation of expressions.

EXPZEROFF, Argument N is invalid. Assembler expected a general register expression of the form 0(Rn) or (Rn).

Error: The cited argument is invalid. The assembler expected a general register expression of the form 0(Rn).

User Action: Check your source code and see Chapter 2 for information about general register expressions.

FOUNDEXP, Assembler found XXXXX when expecting one of the following: XXXXX.

Error: The assembler found an unexpected item in a location where it expected something else.

User Action: Check the unexpected item found in the source statement. Examine those items cited as *expected* as alternatives for the unexpected item.

FREGDEF, You cannot define a floating-point register in terms of an integer register.

Warning: You are attempting to define a floating-point register symbol in terms of an integer register.

User Action: Specify either a floating-point register or an expression within the range of 0 to 31 with the .DEFINE_FREG directive. See Chapter 5 for information about the .DEFINE_FREG directive.

GENERROR, Generated ERROR:

Error: This statement was generated using the .ERROR directive.

User Action: Examine your source code.

GENPRINT, Generated PRINT:

Informational: This statement was generated using the .PRINT directive.

User Action: Examine your source code.

GENWARN, Generated WARNING:

Warning: This statement was generated using the .WARNING directive.

User Action: Examine your source code.

HEXSTR, Illegal hexadecimal escape sequence in string literal.

Error: The specified hexadecimal escape sequence is invalid.

User Action: Check your source code and the documentation.

IDENTTRUNC, The string length of the module IDENT is greater than 31 characters. It is truncated to 31 characters.

Warning: The string argument you specified with the .IDENT directive is too long.

User Action: Specify a shorter string argument. See Chapter 5 for information about the .IDENT directive.

IDFOUND, Assembler found identifier in the opcode field when expecting one of the following: opcode, directive, macro invocation, or symbol definition.

Error: The identifier cited was unexpected. The assembler expected either an opcode, a directive, a macro invocation, or a symbol definition.

User Action: Check your source code.

IDTOOLONG, Identifier is longer than 31 characters.

Error: The identifier exceeds the 31 character maximum size.

User Action: Check your source code and either rename or truncate the identifier.

ILLASCII, Illegal ASCII constant.

Error: The assembler found an illegal ASCII constant with the 6A or ^a radix specifier.

User Action: Check your source code.

ILLBIN, Illegal binary constant.

Error: The assembler found an illegal binary constant with the ^B or ^b radix specifier.

User Action: Check your source code.

ILLDEC, Illegal decimal constant.

Error: The assembler found an illegal binary constant with the ^D or ^d radix specifier.

User Action: Check your source code.

ILLEXPON, Illegal exponent in floating-point constant.

Error: The specified exponent of the floating-point constant is illegal.

User Action: Check your source code and see Chapter 5 for information about floating-point constants.

ILLFLOAT, Illegal floating-point constant.

Error: The specified floating-point constant is illegal.

User Action: Check your source code and see Chapter 5 for information about floating-point constants.

ILLHEX, Illegal hexadecimal constant.

Error: The assembler found an illegal binary constant with the ^X or ^x radix specifier.

User Action: Check your source code.

ILLIFOP, Illegal .IF operator.

Error: An illegal operator was encountered as an .IF operator.

User Action: Check your source code. See Chapter 5 for information about the .IF directive.

ILLINCL, Illegal .INCLUDE file specification.

Error: The assembler encountered an illegal .INCLUDE directive.

User Action: Check your source code. See Chapter 5 for information about the .INCLUDE directive.

ILLOCT, Illegal octal constant.

Error: The assembler found an illegal binary constant with the ^O or ^o radix specifier.

User Action: Check your source code.

ILLOPERANDMIX, Illegal operand mixing for operator.

Error: The resolved operand types are invalid when used together with the specified operator.

User Action: See the descriptions of operators, operands, and expressions in Chapter 2.

ILLPROCRET, Illegal procedure return; linkage register (argument 1) must be R31 when software hint (argument 3) is 1.

Error: Illegal procedure return.

User Action: Check the instruction arguments. When argument 3, software hint, is 1, the first argument specifying the linkage register must be R31.

ILLRADIX, Illegal radix specifier in numeric constant; specify A, B, C, D, O, or X.

Error: The assembler found an illegal radix specifier.

User Action: Check your source code and use one of A, B, C, D, O, or X.

INCLDEPTH, .INCLUDE nest depth exceeds N - check for circular .INCLUDE.

Error: The assembler attempted to exceed the maximum level of include file depth.

User Action: Check your source code for circular file inclusion.

INCLOPEN, .INCLUDE file open error.

Error: The assembler could not open the included file.

User Action: Check the file attributes and so forth of the specified .INCLUDE file.

INSNOTINPSC, Instructions must be in a MIX, NOEXE; MIX, EXE; or NOMIX, EXE PSECT.

Error: You specified an instruction in a psect with incorrect psect attributes.

User Action: Make sure the psect has MIX or EXE and NOMIX attributes set.

INTERNAL, Internal assembler error. Please report the problem to Digital.

Fatal: An internal error has occurred.

User Action: Gather as much information as possible about the circumstances under which the error occurred and report the problem to Digital.

INTERR, Internal processing error in the SYN facility. Please report the problem to Digital.

Fatal: An internal error has occurred.

User Action: Gather as much information as possible about the circumstances under which the error occurred and report the problem to Digital.

INVALIGNFILL, You specified an invalid optional fill pattern with the .ALIGN directive.

Error: You specified an invalid optional fill pattern.

User Action: Check your source code, in particular the second argument to the .ALIGN directive, the alignment fill specifier, to insure that it resolves to an integer. See Chapter 5 for information about the .ALIGN directive.

INVBASEEXP, Invalid expression for .BASE directive.

Error: The expression is not valid for .BASE directive.

User Action: The expression you specified for a base register with the .BASE directive should contain no forward references and resolve to one of the following at this point in assembly: psect +/ − offset, external symbol reference +/− offset, integer, label +/− offset, where the label is defined in a psect with the EXE and NOMIX attributes. See Chapter 5 for more information about the assembler evaluation of expressions.

INVBASEREG, Invalid base register. Base register must be one of R0 through R30.

Error: You specified an invalid base register.

User Action: Specify a base register as a general register from the range of R0 . . . R30. R31 cannot be specified as a base register and is implicitly defined as .BASE R31, 0.

INVBRTGT, Invalid branch target. Branch target label must be defined in same psect as the branch instruction which references the label.

Error: The specified label you reference as the target of a branch instruction must be defined in the same psect in which it is referenced.

User Action: See Chapter 4 for more information about labels.

INVCA, You specified an invalid code address with the procedure descriptor. The code address must be a nontemporary label defined in a psect with the EXE or MIX attribute after its use with .PROCEDURE_DESCRIPTOR.

Error: The code address you specified as the second argument to the .PROCEDURE_DESCRIPTOR directive is invalid.

User Action: The code address must be a non-temporary label defined in a psect with the EXE or NOMIX attribute. Check your source code.

INVEXP, Assembler found XXXXX when expecting a valid expression.

Error: The assembler expected one of the following: integer, floating-point constant, identifier, register, period (.), left-angle bracket (<), or unary operator.

User Action: Check the unexpected item found in the source statement.

INVEXPRFORDIR, Invalid expression type for directive.

Error: The assembler resolved value for the expression in the cited directive is invalid.

User Action: See Chapters 1 and 5 for more information about the directive arguments and types.

INVEXPRFORSYM, Invalid expression type for symbol.

Error: The assembler resolved value for the expression that is assigned to a local or global symbol is invalid.

User Action: Expressions assigned to a symbol must contain no forward references and must resolve to an integer or psect/label +/− offset. See Chapter 2 for more information about how the assembler determines symbol and expression values.

INVFPCONST, Invalid floating-point value. Check value range for floating-point data type.

Error: The assembler detected an invalid floating-point value.

User Action: Check the specified range for the directive type.

INVINSQUAL, You specified an invalid instruction qualifier list for the opcode.

Error: The instruction qualifier you specified with the opcode is invalid.

User Action: See Appendix A for a complete list of opcodes and valid instruction qualifiers.

INVLCA, Assembler found an invalid or undefined code address for the procedure descriptor.

Error: An invalid or undefined code address corresponds to the specified procedure descriptor.

User Action: Check your source code for the specified code address.

INVLISTOPT, You specified an invalid option with the .LIST or .SHOW directive.

Error: You specified an invalid option with the .LIST or .SHOW directive.

User Action: See Chapter 5 for valid .LIST and .SHOW directive options.

INVLPD, Invalid procedure descriptor.

Error: You specified an invalid procedure descriptor. There was no definition of a procedure descriptor by the specified name.

User Action: Check your source code.

INVNLISTOPT, You specified an invalid option with the .NLIST or .NOSHOW directive.

Error: You specified an invalid option with the .NLIST or .NO_SHOW directive.

User Action: See Chapter 5 for valid .NLIST and .NO_SHOW directive options.

INVOFF, You attempted to specify data intialization with a current psect offset that is outside the range of 0 to 2147483647.

Error: The current psect offset is invalid for specifying a data initialization.

User Action: Check your source code and the value of the current psect offset.

INVREGNUMEXP, Invalid register-number expression. Specify an integer expression between 0 and 31 or a previously defined or predefined register.

Error: You specified an illegal expression for a register symbol definition.

User Action: Specify a value between 0 and 31. You can also define a register in terms of a previously-defined or predefined register.

INVREPCOUNT, The integer value of the .REPEAT expression is not within the inclusive range of 0 . . . 65535. A 0 value is assumed.

Warning: The value of the .REPEAT expression must be within the range of 0 . . . 65,535, inclusive. Therefore, a 0 expression value is assumed.

User Action: Specify a repetition count between 0 and 65,535, inclusive.

INVSAVEOPT, You specified an invalid option with the .SAVE_PSECT directive.

Error: You spoecified an invalid option with the .SAVE_PSECT directive.

User Action: See Chapter 5 for valid .SAVE_PSECT directive options.

INVTEMPLAB, Invalid use of temporary label.

Error: A temporary label reference is not allowed in this context.

User Action: See Chapter 2 for information about using temporary labels.

INVTERM, Assembler found N when expecting a valid expression term.

Error: The assembler found an unexpected item where it expected one of the following expressions: floating-point number, integer, register, decimal point (.), identifier, or left-angle bracket (<).

User Action: Check the item flagged by the assembler.

IREGDEF, You cannot define an integer register in terms of a floating-point register.

Warning: You are attempting to define an integer register symbol in terms of a floating-point register. IREGDEF

User Action: Specify either an integer register or an expression within the range of 0 to 31 with the .DEFINE_IREG directive.

LABELNOTDEF, Undefined label.

Error: The label you specified is undefined.

User Action: See Chapters 2 and 4 for descriptions of the valid labels.

LABELREDECL, Illegal redefinition of label.

Error: You have illegally defined this label in multiple places in this assembly unit.

User Action: Check all references to this label in your source code.

LABNOTINPSECT, Label must be declared in a PSECT.

Error: You are attempting to declare a temporary, local, or global label without first establishing a psect.

User Action: Make sure you enter the appropriate .PSECT directive before declaring the label in your source stream.

LEXOPEDITSPEC, Unrecognized edit specifier.

Error: The assembler does not recognize the edit specifier for the %EDIT lexical operator.

User Action: Check your source code and see Chapter 3 for information about the %EDIT lexical operator.

LEXOPENDM, Illegal modification of .ENDM directive keyword by lexical operation.

Error: While your macro definition contains a .ENDM directive that ends the macro definition, the .ENDM directive is modified by a lexical operator so that it can no longer be recognized as a .ENDM directive keyword after lexical processing.

User Action: Change the statement to avoid modifying the .ENDM directive keyword with lexical operator processing. See Chapter 3 for information about using lexical operators.

LEXOPENDR, Illegal modification of .ENDR directive keyword by lexical operation.

Error: While your repeat range contains a .ENDR directive that ends the repeat block, the .ENDR directive is modified by a lexical operator so that it can no longer be recognized as a .ENDR directive keyword after lexical processing.

User Action: Change the statement to avoid modifying the .ENDR directive keyword with lexical operator processing. See Chapter 3 for information about using lexical operators.

LEXOPSYNTAX, Illegal lexical operator syntax (missing left or right parenthesis, missing comma, or other lexical operator syntax error).

Error: The indicated lexical operator has a syntax error.

User Action: Check the source code to insure correct syntax.

LEXSYM, XXXXX is already a lexical string symbol name; it cannot also be a numeric symbol name.

Error: You cannot define a lexical string symbol and a numeric symbol by the same name.

User Action: Check your source code and remove either the lexical string or the numeric symbol definition.

LIBMOD_BADFORMAT, Library module XXXXX contains illegal syntax (missing .MACRO or label preceding .MACRO, missing or not matching .ENDM, or other macro syntax error).

Error: The assembler encountered illegal syntax.

User Action: Check the syntax of the macro.

LIBMOD_EMPTY, Library module XXXXX is empty.

Warning: The assembler encountered an empty library module.

User Action: Replace the library module.

LIBMOD_EXTRA, Library module XXXXX contains extraneous text after .ENDM; the assembler ignores the extra text.

Warning: The assembler encountered extraneous text after an .ENDM directive in a library module. The assembler ignores this text.

User Action: Correct the library module.

LIBMOD_NOT_FOUND, Library module XXXXX not found.

Error: The assembler could not find the indicated library module.

User Action: Check the spelling of the macro library and module names.

LOCCTRNOTDATA, Location counter cannot be set in a psect with the EXE and NOMIX attributes.

Error: You cannot modify the location counter in a psect with the EXE and NOMIX attributes.

User Action: If you need to modify the location counter, specify the MIX psect attribute. See Chapter 5 for a description of the MIX psect attribute.

MACCASEMATCH, Library macro name is spelled using different alphabetic case than in .MCALL directive or macro invocation.

Error: There is an alphabetic case difference between that specified in the macro library and what you specified for the macro name.

User Action: Check the case of the macro name in your source code and the case of the macro in the specified macro library.

MACEXPNEST, Macro expansion exceeds maximum nesting depth (macro recursion not detected).

Error: The macro is not recursive but exceeds the maximum allowable expansion depth.

User Action: Check your source code for possible restructuring.

MACPARAMGENDEF, You can specify a generated label default value or a default string value, but not both.

Error: You specified both a default string value and a generated label default value when you can only specify one.

User Action: Examine your source code.

MACPARAMSYNTAX, Illegal macro parameter syntax. Assembler found XXXXX when expecting one of XXXXX.

Error: Macro parameter syntax is invalid.

User Action: Try replacing the unexpected argument with one of those items cited as *expected*.

MACRECURSE, Recursive macro exceeds maximum macro expansion nesting depth.

Error: The macro is recursive and exceeds the maximum expansion nesting depth.

User Action: Check your source code for a missing basis step in the recursive macro.

MACZERO, Assembler cannot evaluate expression. A 0 expression value is assumed.

Informational: The assembler cannot evaluate this expression due to errors it encountered. Therefore, a 0 value is assumed.

User Action: Check the expression for forward or external references.

MAXIF, Maximum nesting of .IF directives exceeded.

Error: The maximum depth nesting of .IF directives has been exceeded.

User Action: Check your source code for possible restructuring.

MAXLEXOP, More than N lexical operators encountered; check for a recursive lexical string symbol.

Error: Your source line contains an excessive number of lexical operators. A recursive lexical string symbol definition occurs when you initially define a lexical string symbol in terms of itself using the lexical substitution operator. While the assembler normally interprets lexical substitution operators during lexical string symbol definition, it cannot in this case because the lexical string symbol is not yet defined. When the assembler later expands such

a lexical string symbol with the imbedded self reference, infinite recursion results.

User Action: Check your source code for recursive lexical string symbol definitions and redefine them to avoid recursion.

MISSENDC, Missing .ENDC directive(s).

Warning: The assembler could not find a terminating .ENDC conditional directive.

User Action: Check your source code.

MISSINGENDM, Missing .ENDM directive.

Error: The assembler could not find a terminating .ENDM directive to match a .MACRO directive.

User Action: Check your source code.

MISSINGENDR, Missing .ENDR directive.

Error: The assembler could not find an .ENDR directive to terminate an .IRP or a.REPEAT block.

User Action: Check your source code.

MISSQUOTE, Missing closing double-quote character in string literal.

Error: The closing double-quote is missing from the string literal.

User Action: Check your source code and insert a closing double-quote for a string literal.

MODCODLOCCTR, Restoring the location counter in the current context causes an illegal modification of the location counter for a psect with the EXE and NOMIX attributes.

Error: You cannot modify the location counter in a psect with the EXE and NOMIX attributes.

User Action: If you need to modify the location counter, specify the MIX psect attribute. See Chapter 5 for a description of the MIX psect attribute.

NOBEGEX, Assembler encountered an unmatched .END_EXACT directive.

Error: The assembler encountered an .END_EXACT directive before a .BEGIN_EXACT directive.

User Action: Check your source code.

NOCA, You did not specify a code address as argument 2 with .PROCEDURE_DESCRIPTOR.

Error: You did not specify a code address as the second argument to the .PROCEDURE_DESCRIPTOR directive.

User Action: The code address must be a nontemporary label defined in a psect with the EXE or NOMIX attribute. Check your source code.

NOQUAL, Instruction qualifiers are invalid with this opcode.

Error: You cannot specify instruction qualifiers with this opcode.

User Action: See Appendix A for a complete list of opcodes and valid instruction qualifiers.

NOTAQUAL, An item you specified in the qualifier list is invalid with this opcode.

Error: The instruction qualifier you specified with the opcode is invalid.

User Action: See Appendix A for a complete list of opcodes and valid instruction qualifiers.

NOTENOUGHARGS, Not enough arguments for instruction.

Error: The instruction needs one or more additional arguments.

User Action: Check the argument numbers and required formats as specified in Chapter 5.

NOTINMACRO, This statement must occur only within a macro.

Error: The statement you specified is only allowed within a macro.

User Action: See Chapter 2 for a description of the statement specified.

NOTINSEM, Missing functionality in SEM.

Error: This functionality is missing in the assembler.

User Action: Please report the problem to Digital.

NUMSYM, XXXXX is already a numeric symbol name; it cannot also be a lexical string symbol name.

Error: You cannot define a numeric symbol and a lexical string symbol by the same name.

User Action: Check your source code and remove either the numeric or the lexical string symbol definition.

OPTIGN, The assembler is ignoring one or more VAX MACRO options.

Informational: The assembler detected and ignored a VAX MACRO option.

User Action: Remove the VAX MACRO options from your MACRO–64 Assembler for OpenVMS AXP Systems program.

OVERLAP, Assembler detected overlapping initializers at offset NN. This initial value overlaps but is not an exact replacement for a previous initial value.

Error: You are trying to assign or initialize multiple values to the same location. This is not allowed.

User Action: Check your source code.

PSECTALIGNCON, PSECT alignment conflicts with earlier declaration.

Error: A previously specified psect alignment attribute conflicts with the flagged psect attribute.

User Action: Check all declarations of the psect. See Chapter 5 for a description of the .PSECT directive and psect attributes.

PSECTATTRCON, PSECT attribute conflicts with earlier declaration.

Error: A previously specified psect attribute conflicts with the flagged psect attribute.

User Action: Check all declarations of the psect. See Chapter 5 for a description of the .PSECT directive and psect attributes.

REGREDEF, You attempted to redefine a previously defined register symbol with a different value.

Warning: You are attempting to change the definition of a register symbol that either you have previously defined or that MACRO–64 has predefined.

User Action: Check for conflicts with the register-symbol identifier you have specified. If you wish to redefine a register symbol, you must first cancel its previous definition with the .UNDEFINE_REG directive.

REDUNDELSE, You cannot specify more than one .ELSE directive within a single .IF block.

Error: The assembler encountered more than one .ELSE directive within a single .IF block.

User Action: Check your source code.

RESTOREWOSAVE, PSECT .RESTORE without .SAVE.

Error: You entered a .RESTORE_PSECT directive without a previous corresponding .SAVE_PSECT directive.

User Action: Check the uses of .SAVE_PSECT and .RESTORE_PSECT in your source code.

SAVESTACKOVER, Internal SEM PSECT .SAVE stack overflow.

Fatal: An internal error has occurred.

User Action: Gather as much information as possible about the circumstances under which the error occurred and report the problem to Digital.

SRCREAD, Error reading source file.

Error: The assembler encountered an error in reading your source file.

User Action: Check file specifications, protections, and so forth.

SYMBOLREDECL, Illegal redefinition of symbol.

Error: The symbol is already defined as a label or explicitly declared as externally defined with the .EXTERNAL directive.

User Action: Check all uses of this symbol.

TOOMANYARGS, Too many arguments for instruction.

Error: The instruction contains one or more arguments than necessary.

User Action: Check the argument numbers and required formats as specified in Chapter 5.

TOOMANYMACARG, You specified more arguments than are defined for this macro.

Error: You specified more arguments on the macro call than were specified for its definition.

User Action: Check the macro definition and point of call in your source code.

TOOMANYMACPARAMS, Too many macro parameters.

Error: You specified too many macro parameters.

User Action: Check your source code.

TRUNCDATA, Data truncation warning.

Warning: You specified a data value that is out of range for the specified directive, which results in data truncation.

User Action: Specify a smaller value.

UNDCA, You specified an undefined code address with the procedure descriptor.

Error: The code address you specified as the second argument to the .PROCEDURE_DESCRIPTOR directive is undefined.

User Action: The code address must be a nontemporary label defined in a psect with the EXE or NOMIX attribute. Check your source code.

UNDEFSYM, Undefined symbol or label. Assembler assumes an .EXTERNAL definition.

Warning: The referenced label or symbol does not have an explicit definition and an external definition is assumed.

User Action: Use the .EXTERNAL directive to declare the symbol.

UNEXPELSE, Unexpected .ELSE directive.

Error: The assembler encountered an unexpected .ELSE directive.

User Action: Check the use of the .ELSE directive in your source code to insure proper positioning with a .IF and .ENDC directive.

UNEXPENDC, Unexpected .ENDC directive.

Error: The assembler could not find a terminating .ENDC for a macro conditional directive, such as .IF.

User Action: Check your source code.

UNEXPENDM, Unexpected .ENDM directive.

Error: The assembler encountered an unexpected .ENDM directive.

User Action: Check your source code for matching .MACRO/.ENDM pairs.

UNEXPENDR, Unexpected .ENDR directive.

Error: The assembler encountered an unexpected .ENDR directive.

User Action: Check your source code for matching .REPEAT/.ENDR and .IRP/.ENDR directive pairs.

UNEXPIFF, Unexpected .IF_FALSE (.IFF) directive.

Error: The assembler encountered an unexpected .IF_FALSE directive.

User Action: Check your source code to insure that this directive occurs within an .IF block.

UNEXPIFT, Unexpected .IF_TRUE (.IFT) directive.

Error: The assembler encountered an unexpected .IF_TRUE directive.

User Action: Check your source code to insure that this directive occurs within an .IF block.

UNEXPIFTF, Unexpected .IF_TRUE_FALSE (.IFTF) directive.

Error: The assembler encountered an unexpected .IF_TRUE_FALSE directive.

User Action: Check your source code to insure that this directive occurs within an .IF block.

UNEXPMEXIT, Unexpected .MEXIT directive.

Error: The assembler encountered an unexpected .MEXIT directive.

User Action: Check your source code.

UNKDIR, Assembler found unknown directive XXXXX.

Error: An internal error has occured.

User Action: Gather as much information as possible about the circumstances under which the error occurred and report the problem to Digital.

UNKENDISOPTION, You specified an unknown .ENABLE/.DISABLE option.

Error: The option you specified for .ENABLE/.DISABLE is incorrect.

User Action: Check the option specified with the .ENABLE/.DISABLE directive.

UNKNOWNATTR, Unknown PSECT attribute.

Error: The assembler does not recognize the specified psect attribute.

User Action: See Chapter 5 for a description of the .PSECT directive and psect attributes.

UNTERMEX, Assembler detected N unterminated .BEGIN_EXACT directive(s) in psect XXXXX.

Error: Unmatched .BEGIN_EXACT directive(s) occur for the indicated psect.

User Action: Check your source code.

VAXDIR, The assembler is ignoring one or more VAX MACRO directives or options. Assembler continues processing with the next line.

Informational: The assembler detected and ignored a VAX MACRO directive or option.

User Action: Remove the VAX MACRO directives from your MACRO–64 Assembler for OpenVMS AXP Systems program.

VMACELSE, You cannot specify .ELSE in the same .IF block with either .IF_ FALSE (.IFF), .IF_TRUE (.IFT), or .IF_TRUE_FALSE (.IFTF).

Error: The assembler encountered an .ELSE directive within the same .IF block as an .IF_FALSE, .IF_TRUE, or .IF_FALSE directive.

User Action: Check your source code and remove either the .ELSE directive or the .IF_x directive.

WRONGMACID, Macro name in .ENDM does not match corresponding .MACRO.

Error: The macro name you specified as the optional argument to the .ENDM directive does not match the name you specified with the corresponding .MACRO directive.

User Action: Check your souce code for matching .MACRO/.ENDM directive pairs.

WRONGPL, The code address you specify with the .PROCEDURE_ DESCRIPTOR directive must occur BEFORE its definition as a local or global label.

Error: The code address you specify as the second argument to the .PROCEDURE_DESCRIPTOR directive must occur before its definition as a nontemporary label defined in a psect with the EXE or NOMIX attribute.

User Action: Check your source code.

F

MACRO–64 Listing Format

This appendix contains a sample MACRO–64 program, comprised of Examples F–1 and F–2, that shows the listing file format. Example F–3 shows the resulting output file in binary listing format.

For more information about control of the listing file and its contents, see the description of the .SHOW directive in Chapter 5.

Example F–1 Main Source File

```
.title LIST "Example"
.ident "V1.0"
.show include, conditionals
.show expansions
.enable align_data
.enable peephole, schedule

.subtitle "Definitions"
.include "DEFINITIONS.M64"

.noshow expansions

.subtitle "Linkage"
$linkage_section
DATA_PTR:
.address DEFAULT_DATA
.if eq, OPTION, 1
  . = DATA_PTR
  .address OPTIONAL_DATA
.endc

.subtitle "Routine ZING"
.show binary
$routine ZING, kind=stack
  .base r27,$ls
  addq  r16,r17,r22
  addq  r1,r18,r22
  addq  r19,r20,r23
  addq  r21,r22,r23
  ldq   r0,DATA_PTR
  .begin_exact
    stq r31,0(r0)
    stq r22,8(r0)
    stq r23,16(r0)
  .end_exact
  $return
$end_routine ZING

.end
```

Example F–2 Include Source File

```
.macro DEFINE SYMBOL, VALUE, SOFT=FALSE
  .if identical, <SOFT>, <TRUE>
    .if defined, SYMBOL
      .print "Using /DEFINE=SYMBOL=%integer(SYMBOL)"
      .mexit
    .endc
  .endc
  SYMBOL = VALUE
.endm DEFINE

DEFINE OPTION, VALUE=0, SOFT=TRUE
```

Example F–3 Example Listing Output

```
LIST   Example      ❶       26-AUG-1993 11:38:22   MACRO-64 V1.1   Page   1
V1.0   Definitions          26-AUG-1993 11:38:17   [PROJ]EXAMPLE.M64;3

                          ❷ 1 .title LIST "Example"
                            2 .ident "V1.0"
                            3 .show include, conditionals ❸
                            4 .show expansions
                            5 .enable align_data
                            6 .enable peephole, schedule
                            7
                            8 .subtitle "Definitions"
                            9 .include "DEFINITIONS.M64"
                     ❹ 1   10 .macro DEFINE SYMBOL, VALUE, SOFT=FALSE
                       1   11   .if identical, <SOFT>, <TRUE>
                       1   12     .if defined, SYMBOL
                       1   13       .print "Using /DEFINE=SYMBOL=%integer(
                                  SYMBOL)" ❺
                       1   14       .mexit
                       1   15     .endc
                       1   16   .endc
                       1   17   SYMBOL = VALUE
                       1   18 .endm DEFINE
                       1   19
                       1   20 DEFINE OPTION, VALUE=0, SOFT=TRUE
                              1
%MACRO64-I-GENPRINT, (1) Generated PRINT: Using /DEFINE=OPTION=1 ❻

%MACRO64-I-MACAUXMSG1, (1) This condition occurred in the expansion of macro DEF
INE at line 3 of its definition.
\      .print "Using /DEFINE=OPTION=1"\
```

(continued on next page)

Example F–3 (Cont.) Example Listing Output

```
                    ❼ 1  1           .if identical, <TRUE>, <TRUE>
                      1  1             .if defined, OPTION
                      1  1               .print "Using /DEFINE=OPTION=%integer(
                                        OPTION)"
*INFO* ❽              1  1* ❾            .print "Using /DEFINE=OPTION=1"
                      1  1               .mexit
                                  21
                                  22 .noshow expansions ❿
                                  23
                                  24 .subtitle "Linkage"
                                  25 $linkage_section
                     0000 ⓫       757 DATA_PTR:
XXXXXXXXXXXXXXXX 0000 ⓬           758 .address DEFAULT_DATA
0000000000000001 0008 ⓭           759 .if eq, OPTION, 1
0000000000000001 0008
0000000000000000'0000 ⓮           760   . = DATA_PTR
0000000000000000'0000             761   .address OPTIONAL_DATA
                     0008         762 .endc
                     0008         763
                     0008         764 .subtitle "Routine ZING"
                     0008         765 .show binary ⓯
                     0008         766 $routine ZING, kind=stack
⓰             00000000  2   *          .psect $DATA$,OCTA,NOPIC,CON,REL,LCL,NOSHR
                                      ,NOEXE,RD,WRT
              00000008  2   *          .psect $LINK$,OCTA,NOPIC,CON,REL,LCL,NOSHR
                                      ,NOEXE,RD,NOWRT
             3089 0008  2             .word MACRO64$PD_KIND ! MACRO64$PD_FLAGS
             0008 000A  2             .word $RSA_OFFSET
               00 000C  2             .byte 0
               00 000D  2             .byte <MACRO64$FUNC_RETURN&^X1F>!<<MACRO64
                                      $EXCEPTION_MODE&^X1F>@4>
             0001 000E  2             .word MACRO64$DEFAULT_SIGNATURE
0000000000000000'0010  2             .address <MACRO64$ENTRY_0>
```

(continued on next page)

Example F–3 (Cont.) Example Listing Output

```
LIST   Example              26-AUG-1993 11:38:22   MACRO-64 V1.1   Page   2
V1.0   Routine ZING         26-AUG-1993 01:29:34   [PROJ]EXAMPLE.M64;3

        00000020 0018  2             .long $SIZE
            0000 001C  2             .word 0
            0014 001E  2             .word MACRO64$END_PROLOGUE_0-MACRO64$ENTRY
                                    _0
        20000000 0020  2             .long MACRO64$SAVED_IREG_MASK
        00000000 0024  2             .long MACRO64$SAVED_FREG_MASK
             00000000  2   *         .psect $CODE$,OCTA,PIC,CON,REL,LCL,SHR,EXE
                                    ,NORD,NOWRT
        23DEFFE0 0000  3             LDA $SP, -$SIZE($SP)
        B77E0000 0004  2             .iif ne, <MACRO64$PD_FLAGS&MACRO64$M_BASE_
                                    REG_IS_FP>, STQ $PV, ($SP)
        B75E0008 0008  2             STQ  $RA, $RSA_OFFSET($SP)
        B7BE0010 000C  3   *         STQ  MACRO64$R29,$RSA_OFFSET+<MACRO64$RS_C
                                    OUNT*8>($SP)
        47FE041D 0010  2             MOV  $SP, $FP
        00000008'0014          2090  .base r27,$ls
      *Optimized* ⓱            2091  addq  r16,r17,r22
      *Optimized*              2092  addq  r1,r18,r22
      *Optimized*              2093  addq  r19,r20,r23
      *Optimized*              2094  addq  r21,r22,r23
      *Optimized*              2095  ldq   r0,DATA_PTR
                               2096  .begin_exact
        B7E00000 0028          2097    stq r31,0(r0)
        B6C00008 002C          2098    stq r22,8(r0)
        B6E00010 0030          2099    stq r23,16(r0)
                               2100  .end_exact
                  0034         2101  $return
      *Optimized*      1             .iif ne, <MACRO64$PD_FLAGS&MACRO64$M_BAS
                                    E_REG_IS_FP>, MOV $FP, $SP
      *Optimized*      1             LDQ  MACRO64$R28, $RSA_OFFSET($SP)
      *Optimized*      1             LDQ  $FP, MACRO64$FP_OFFSET($SP)
      *Optimized*      2             LDA $SP, $SIZE($SP)
      *Optimized*      1             RET  (MACRO64$R28)
                               2205 $end_routine ZING
                  0048         2206
                               2207 .end
```

(continued on next page)

Example F–3 (Cont.) Example Listing Output

```
LIST ⓲Machine Code Listing  26-AUG-1993 11:38:22   MACRO-64 V1.1   Page   3
V1.0   $CODE$               26-AUG-1993 11:38:17   [PROJ]EXAMPLE.M64;3

                                .PSECT  $CODE$, OCTA, PIC, CON, REL, LCL, SHR,-
                                        EXE, NORD, NOWRT

            0000        ZING::
            0000        MACRO64$ENTRY_0:
23DEFFE0 ⓳  0000 ⓴          ㉑ LDA     SP, -32(SP)   ; SP, -32(SP) ㉒ ; 000766
B77E0000    0004            STQ     R27, (SP)     ; R27, (SP)
B75E0008    0008            STQ     R26, 8(SP)    ; R26, 8(SP)
B7BE0010    000C            STQ     FP, 16(SP)    ; FP, 16(SP)
47FE041D    0010            MOV     SP, FP        ; SP, FP
            0014        MACRO64$END_PROLOGUE_0:
42110416    0014            ADDQ    R16, R17, R22 ; R16, R17, R22  ; 002091
A41BFFF8    0018            LDQ     R0, -8(R27)   ; R0, -8(R27)    ; 002095
40320416    001C            ADDQ    R1, R18, R22  ; R1, R18, R22   ; 002092
42740417    0020            ADDQ    R19, R20, R23 ; R19, R20, R23  ; 002093
42B60417    0024            ADDQ    R21, R22, R23 ; R21, R22, R23  ; 002094
B7E00000    0028            STQ     R31, (R0)     ; R31, (R0)      ; 002097
B6C00008    002C            STQ     R22, 8(R0)    ; R22, 8(R0)     ; 002098
B6E00010    0030            STQ     R23, 16(R0)   ; R23, 16(R0)    ; 002099
47FD041E    0034            MOV     FP, SP        ; FP, SP         ; 002101
A79E0008    0038            LDQ     R28, 8(SP)    ; R28, 8(SP)
A7BE0010    003C            LDQ     FP, 16(SP)    ; FP, 16(SP)
23DE0020    0040            LDA     SP, 32(SP)    ; SP, 32(SP)
6BFC8001    0044            RET     R28           ; R28

Routine Size: 72 bytes,    Routine Base: $CODE$ + 0000 ㉓

                                .PSECT  $LINK$, OCTA, NOPIC, CON, REL, LCL,-
                                        NOSHR, NOEXE, RD, NOWRT
00000000    0000            .ADDRESS  OPTIONAL_DATA
    3089    0008            .WORD   X^3089
    0008    000A            .WORD   X^0008
      00    000C            .BYTE   X^00
      00    000D            .BYTE   X^00
    0001    000E            .WORD   X^0001
00000000    0010            .ADDRESS  MACRO64$ENTRY_0
00000020    0018            .LONG   X^00000020
    0000    001C            .WORD   X^0000
    0014    001E            .WORD   X^0014
20000000    0020            .LONG   X^20000000
00000000    0024            .LONG   X^00000000

Command: MACRO/ALPHA/LIST/MACHINE_CODE/DEFINE=OPTION=1 EXAMPLE ㉔
```

❶ Each page of the listing begins with a two-line page header. From left to right, the first header line contains the followings fields:

1. The module name as specified with the first argument to the .TITLE directive.
2. The module's title as specified with the second argument to the .TITLE directive.
3. The date and time of the assembly.
4. The MACRO–64 identification and version string.
5. The page number.

From left to right, the second header line contains the followings fields:

The module identification string, as specified with the .IDENT directive.

The subtitle, as specified with the .SUBTITLE directive.

The date and time of the source file.

The source file specification.

❷ Each source line is shown in the listing to the right of its listing line number. The listing line number is shown in decimal radix. It starts at one and increments for every source line and every macro library line whether or not that line is actually shown in the listing.

❸ The .SHOW directive controls the types of information shown in the listing. The INCLUDE option causes MACRO–64 to show include files (files that are included with the .INCLUDE directive) in the listing. The CONDITIONALS option causes MACRO–64 to show blocks of code in the listing that have been conditionally excluded from the assembly. The EXPANSIONS option causes MACRO–64 to show macro expansion lines in the listing. See Chapter 5 for additional information about the .SHOW directive.

❹ This field shows the include-file nesting level.

❺ With long lines, MACRO–64 shows as many characters as can fit within the 132-column listing width. Additional characters are shown on one or more continuation lines, beginning in the source text field.

❻ MACRO–64 shows diagnostic messages in association with the source code that caused the diagnostic. If the diagnostic occurs in a macro expansion, MACRO–64 shows the primary diagnostic citation in association with the macro invocation statement. In addition, if the BINARY or EXPANSIONS listing option is in effect, MACRO–64 shows a secondary diagnostic citation in association with the macro expansion statement that caused the diagnostic, as shown in callout ❽.

❼ This field shows the macro invocation nesting level.

❽ As described with callout ❻, MACRO–64 shows a secondary diagnostic citation in association with macro expansion lines that cause diagnostics. MACRO–64 shows the primary diagnostic citation in association with the macro invocation statement.

❾ MACRO–64 indicates lines that have been modified by lexical operators by placing an asterisk (*) in this field. MACRO–64 also indicates macro library lines by placing the letter L in this field.

❿ This statement disables the EXPANSIONS option, which causes MACRO–64 to suppress macro expansion lines from the listing.

⓫ This field shows the current psect offset. The .PSECT, .SAVE_PSECT, and .RESTORE_PSECT directives show the full 32-bit offset using eight hexadecimal digits. While MACRO–64 maintains a 32-bit psect offset, other statements show only the low-order 16-bits of the offset using four hexadecimal digits.

⓬ MACRO–64 shows data initial values that have been replaced using a series of Xs where the data value would otherwise have been displayed.

⓭ MACRO–64 shows data initial values and other associated values in the left margin. Quadword data is shown as 16 hexadecimal digits. Longword data is shown as eight hexadecimal digits. Word data is shown as four hexadecimal digits. Byte data is shown as a pair of hexadecimal digits. Octaword data is shown as a pair of quadwords. MACRO–64 places additional lines into the listing to accommodate the display of data in the left margin as necessary.

⓮ MACRO–64 uses the single quote (') to indicate a relocatable value that cannot be bound at assembly time.

⓯ This statement enables the BINARY listing option. It causes MACRO–64 to include in the listing only those macro expansion lines that produce a binary result in the object module. Other macro expansion lines are not included in the listing. Regular source lines are not affected.

⓰ While the $ROUTINE macro expands to several hundred assembly statements, only those that generate a binary result are shown in the listing while the the BINARY option is in effect.

⓱ There is no way for MACRO–64 to correlate the instructions resulting from optimization on a one-to-one basis with your source statements. Instead, MACRO–64 indicates instructions that can be transformed by optimization by displaying the "*Optimized*" string where the instruction hexadecimal representation would otherwise be displayed. MACRO–64 displays the "*Optimized*" string only for those sequences of instructions where optimzation is enabled.

⓲ This is the Machine Code Listing section of the listing. Identical in format to the Machine Code Listing section produced by many of Digital's other AXP language processors, it is useful in viewing the results of optimization.

⓳ This field shows a hexadecimal representation of the instruction or data directive.

⓴ This field shows the low-order 16 bits of the current psect offset using hexadecimal radix.

㉑ This field shows a mnemonic representation of the instruction or data directive.

㉒ This field shows the listing line number that is associated with the instruction or data directive using decimal radix.

㉓ After each routine, MACRO–64 shows the routine size in bytes using decimal radix; it shows the starting offset of the routine's code section using hexadecimal radix.

㉔ At the end of the listing, MACRO–64 echoes the command line used to invoke MACRO–64.

Index

A

Absolute expression, 2–15
Accessing memory, B–4
Address
  base register (.BASE), 5–13
  label, 2–12
  local code storage directive (.LOCAL_CODE_ADDRESS), 5–63
  storage directive (.CODE_ADDRESS), 5–21
.ADDRESS directive, 5–6, B–14
Addressing
  controlling alignment, B–14
  to linkage section, B–12
Address storage directive (.ADDRESS), 5–6
.ALIGN directive, 5–7
/ALIGNMENT qualifier, 1–2
AND operator, 2–19
Architecture instructions, common, A–15 to A–23
ARGS
  arguments, 6–11
  parameters, 6–11
Argument
  actual, 4–1
  determining length, 5–75
  formal, 4–1
  in a macro, 4–1
  number of, 5–74
Arithmetic shift operator, 2–19
.ASCIC directive, 5–9
.ASCID directive, 5–10, B–15
.ASCII directive, 5–11
ASCII string storage directive
  counted (.ASCIC), 5–9
  string (.ASCII), 5–11
  string descriptor (.ASCID), 5–10
  zero-terminated (.ASCIZ), 5–12
.ASCIZ directive, 5–12
Assembler, invoking, 1–2
Assembler directives, 1–1 to 1–2
Assembly termination directive (.END), 5–34
Assignment statement, 1–1, 2–19
Automatic alignment
  controlling, B–13
  default command-line qualifiers, B–13
Automatic alignment directives
  See Table B–2
Automatic data alignment
  label addresses and, 2–13

B

.BASE directive, 5–13, 6–6
Base register, B–4
  defining address, B–12
Base register directive (.BASE), 5–13
$BEGIN_EPILOGUE macro, 6–5, 6–9
.BEGIN_EXACT directive, 5–16
Binary listing file, F–1
Binary operator, 2–18
.BLKA directive, B–15
.BLKD directive, B–15
.BLKF directive, B–15
.BLKG directive, B–15
.BLKL directive, B–15
.BLKO directive, B–15
.BLKQ directive, B–15
.BLKS directive, B–15
.BLKT directive, B–15
.BLKW directive, B–15
.BLK*x* directive, 5–18
Block storage allocation directive (.BLK*x*), 5–18
.BYTE directive, 5–20
Byte storage directive (.BYTE), 5–20

C

$CALL
  calling system services, B–2
Calling-standard macros
  effects on assembly, B–3
Calling system services
  using $CALL, B–2
$CALL macro, 6–6, 6–7, 6–10
  instruction sequence, 6–8
Character set
  in source statement, 2–4
  in source statement (tab.), 2–5
Character string
  determining length, 5–75

$CODE$ symbol, 6–38
.CODE_ADDRESS directive, 5–21, B–14
$CODE_SECTION macro, 6–15
Colon (:)
  in label field, 2–2
Comment field, 2–1, 2–4
Common architecture
  instructions, A–15 to A–23
  opcodes, A–6 to A–11
Compatibility
  with VAX MACRO, D–1 to D–6
Complex expression, 2–15
Conditional assembly block directive
  .ELSE, 5–27
  .ENDC, 5–35
  .IF, 5–45
Created temporary label
  range, 2–11
  using, 4–7
$CS symbol, 6–3, 6–38
Current location counter, 2–20 to 2–21

D

$DATA$ symbol, 6–38
Data alignment
  natural boundaries, B–13
Data Storage
  ALIGN_DATA, 5–28
Data structures, A–1, B–8
  linkage pair, B–9
  procedure descriptor, B–8
  signature block, B–8
$DATA_SECTION macro, 6–16
Date and time lexical
  %TIME, 3–18
/DEBUG qualifier, 1–3
Decodable pseudo-operations, A–6
Default values
  using, 4–2
Define floating point register symbol directive (.DEFINE_FREG), 5–23
Define integer register symbol directive (.DEFINE_IREG), 5–25
/DEFINE qualifier, 1–3
.DEFINE_FREG directive, 5–23
.DEFINE_IREG directive, 5–25
Delimiter
  string argument, 4–3
/DIAGNOSTIC qualifier, 1–4
Differences
  from VAX MACRO, D–1 to D–6
Direct assignment statement, 1–1, 2–19
Directives, 1–1, 5–1
  as operators, 2–3
  for auto alignment, B–14
  general assembler, 1–1, 5–1, 5–3
  macro, 1–2, 5–1, 5–4
Disable assembler functions directive (.DISABLE), 5–26
.DISABLE directive, 5–26
Displaying diagnostic messages
  with LSE, C–1
Displaying error messages
  with LSE, C–1
.DOUBLE directive, 5–22
Double-precision IEEE directive (.T_FLOATING), 5–99
$DP symbol, 6–3, 6–38
.DSABL directive, 5–26
$DS symbol, 6–3, 6–38
.D_FLOATING directive, 5–22, B–14

E

%EDIT lexical, 3–8
Element extraction lexical
  %ELEMENT, 3–9
%ELEMENT lexical, 3–9
.ELSE directive, 5–27
.ENABL directive, 5–28
Enable assembler functions, 5–28
.ENABLE directive, 5–28
.ENDC directive, 5–35
End conditional assembly directive (.END), 5–35
.END directive, 5–34
End macro definition directive (.ENDM), 5–36
.ENDM directive, 5–36
.ENDR directive, 5–37
$END_EPILOGUE macro, 6–5, 6–17
.END_EXACT directive, 5–38
$END_PROLOGUE macro, 6–4, 6–18
$END_ROUTINE macro, 6–19
/ENVIRONMENT qualifier, 1–4
Epilogue sequence, 6–30, B–7, B–11
  beginning, 6–9
.ERROR directive, 5–39
Error messages, E–1 to E–20
.EVEN directive, 5–40
Exact instruction block
  beginning (.BEGIN_EXACT), 5–16
  ending (.END_EXACT), 5–38
Exclusive OR operator, 2–19
Expression, 2–15
  absolute, 2–15
  complex, 2–15
  evaluation of, 2–15
  external, 2–15
  global, 2–15
  relocatable, 2–15, 2–21
  unary operators for, 2–17
.EXTERNAL directive, 5–41
External expression, 2–15
External symbol, 5–103
  attribute directive (.EXTERNAL), 5–41
  defining, 5–28, 5–41

%EXTRACT lexical, 3–10
.EXTRN directive, 5–41

F

Fields
comment, 2–1, 2–4
label, 2–1, 2–2
operand, 2–1, 2–3
operator, 2–1, 2–3
.FLOAT directive, 5–42
Floating-point arithmetic directive
.S_FLOATING, 5–93
.T_FLOATING, 5–99
Floating-point constants (.D_FLOATING), 5–22
Floating-point number
arithmetic, 5–93, 5–99
data structures, A–1
format, 2–6
.F_FLOATING, 5–42
.G_FLOATING, 5–43
in source statement, 2–6
storage, 5–22
storing, 5–42, 5–43
.S_FLOATING, 5–93
.T_FLOATING, 5–99
Floating-point register number lexical
%FREG, 3–11
Floating-point storage directive
.D_FLOATING, 5–22
.F_FLOATING, 5–42
.G_FLOATING, 5–43
Formal argument, 4–1
F-P control register format, A–5
%FREG lexical, 3–11
FUNC_RETURN argument, 6–12
.F_FLOATING directive, 5–42, B–14

G

General assembler directives, 1–1
Global expression, 2–15
Global label, 2–2
Global symbol, 2–10, 5–103
defining, 5–28
.G_FLOATING directive, 5–43, B–14

I

.IDENT directive, 5–44
Identification directive (.IDENT), 5–44
.IF directive, 5–45
.IF_FALSE directive, 5–49
.IF_TRUE directive, 5–49
.IF_TRUE_FALSE directive, 5–49
.IF_*x* directive, 5–49
.IIF directive, 5–52
Immediate conditional assembly block directive (.IIF), 5–52
.INCLUDE directive, 5–53
Inclusive OR operator, 2–19
Indefinite repeat argument directive (.IRP), 5–55
Indefinite repeat character directive (.IRPC), 5–57
Inlining
manual, B–16
optimization, B–16
Instruction directive, 5–54
.INSTRUCTION directive, 5–54, B–15
Instruction formats (fig.), A–1
Instruction operand notation, A–4
Instruction qualifier notation, A–5
Instructions
as operators, 2–3
common architecture, A–15 to A–23
Instruction summary
OpenVMS PALcode, A–11 to A–15
Integer
data structures, A–1
in source statement, 2–6
Integer division
longword, B–2
quadword, B–2
%INTEGER lexical, 3–12
Integer register number lexical
%IREG, 3–13
Invoking assembler, 1–2
%IREG lexical, 3–13
.IRPC directive, 5–57
.IRP directive, 5–55

K

Keyword argument, 4–2
KIND argument, 6–4, 6–5

L

Label
created temporary, 4–7
global, 2–2
local, 2–3
temporary, 2–3
user-defined temporary, 2–11, 4–7
Label addresses, 2–12 to 2–14
Label field, 2–1, 2–2
Label field (tab.), 2–1
Labeling directive
.LOCAL_PROCEDURE_DESCRIPTOR, 5–65
.PROCEDURE_DESCRIPTOR, 5–83
Length Determining lexical
%LENGTH, 3–14
%LENGTH lexical, 3–14

%%—Lexical escape operator, 3–3
Lexical escape operator (%%), 3–3
Lexical operators, 3–1
escape, 3–3
list, 3–7
processing, 3–1
summary of (tab.), 3–7
syntax rules, 3–1, 3–2
using, 3–5
Lexical operators (tab.), 3–7
Lexical string symbol, 3–3
with quoted string literal, 3–3
.LIBRARY directive, 5–59
/LIBRARY qualifier, 1–4
$LINK$ symbol, 6–38
.LINKAGE_PAIR directive, 5–61, B–14
$LINKAGE_PAIR macro, 6–20
$LINKAGE_SECTION macro, 6–21
Linking directive (.LINKAGE_PAIR), 5–61
.LIST directive, 5–62
Listing control directive
.IDENT, 5–44
.LIST, 5–62
.NLIST, 5–76
.NOSHOW, 5–77
Listing directive (.SHOW, NOSHOW), 5–77, 5–94
Listing format
binary, F–1
Listing table of contents, 5–98
/LIST qualifier, 1–4
Local address storage directive (.LOCAL_CODE_ADDRESS), 5–63
LOCAL argument, 6–7, 6–11
Local label, 2–3
saving, 5–92
Local label block
ending, 5–28
starting, 5–28
Local linking directive (.LOCAL_LINKAGE_PAIR), 5–64
Local symbol, 2–10
.LOCAL_CODE_ADDRESS directive, 5–63, B–14
.LOCAL_LINKAGE_PAIR directive, 5–64, B–14
.LOCAL_PROCEDURE_DESCRIPTOR directive, 5–65, B–14
%LOCATE lexical, 3–15
Location control directive
.ALIGN, 5–7
.BLK*x*, 5–18
Location counter
assigning before alignment, 2–13
changing
with EXE and NOMIX attributes, 2–21
with EXE or NOMIX attributes, 2–21
current, 2–20
setting, 2–21
Location counter alignment directive (.ODD), 5–79
Location counter control directive (.EVEN), 5–40
Logical AND operator
See AND operator
Logical exclusive OR operator
See Exclusive OR operator
Logical inclusive OR operator
See Inclusive OR operator
.LONG directive, 5–66, B–14
Longword storage directive (.LONG), 5–66
Loop unrolling
manual, B–16
optimization, B–16
with scheduling and peephole optimizations, B–16
LS argument, 6–10
LSE
displaying diagnostic messages, C–1
displaying error messages, C–1
replicating OpenVMS VAX operating system behavior, C–1
using with MACRO–64, C–1
$LS symbol, 6–3, 6–7, 6–38

M

/MACHINE_CODE qualifier, 1–5
MACRO–64 Assembler for OpenVMS AXP Systems
introduction, 1–1
MACRO–64 features
not in VAX MACRO
additional lexical operators, D–2
automatic data alignment, D–3
.BASE directive, D–1
.BEGIN_EXACT directive, D–3
.CODE_ADDRESS directive, D–1
.ELSE directive, D–2
.END_EXACT directive, D–3
.INCLUDE directive, D–2
.INSTRUCTION directive, D–3
lexical escape operator, D–2
lexical string symbols, D–2
lexical substitution operator, D–2
.LINKAGE_PAIR directive, D–1
.LOCAL_CODE_ADDRESS directive, D–1
.LOCAL_LINKAGE_PAIR directive, D–1
.LOCAL_PROCEDURE_DESCRIPTOR directive, D–1
MACRO64$ symbol, D–3
/[NO]DEFINE qualifier, D–3
$OPDEF macro, D–3
optimizations, D–3
optional case sensitivity, D–2
preprocessor output, D–3
.PROCEDURE_DESCRIPTOR directive, D–1

MACRO–64 features
not in VAX MACRO (cont'd)
register symbol changes, D–4
value-of-operator changes, D–2
Macro arguments, 4–1
actual, 4–1
ARGS, 6–6, 6–11
ARGS parameters, 6–11
concatenated, 4–6
delimited, 4–3, 4–5
formal, 4–1
FUNC_RETURN, 6–12
keyword, 4–2
KIND, 6–4, 6–5
LOCAL, 6–7, 6–11
LS, 6–10
NAMES, 6–8
NONSTANDARD, 6–13
positional, 4–2
Rls, 6–6, 6–10
RSA_END, 6–4
RSA_OFFSET, 6–4
SAVED_REGS, 6–5
SCRATCH_REGS, 6–8, 6–12
SET_ARG_INFO, 6–11
SIGNATURE_BLOCK, 6–13
SIZE, 6–3, 6–5
STACK_RETURN_VALUE, 6–12
STANDARD_PROLOGUE, 6–4
string, 4–3
TIE, 6–12
USES_VAX_ARGLIST, 6–13
Macro call directive (.MCALL), 5–70
Macro calls, 4–1
as operators, 2–3
nested, 5–68
number of arguments, 5–74
MACRO command qualifiers, 1–2
/ALIGNMENT qualifier, 1–2
/DEBUG, 1–3
/DEFINE, 1–3
/DIAGNOSTIC, 1–4
/ENVIRONMENT, 1–4
/LIBRARY, 1–4
/LIST, 1–4
/MACHINE_CODE, 1–5
/NAMES, 1–5
/OBJECT, 1–5
/OPTIMIZE, 1–5
/PREPROCESSOR_ONLY, 1–6
/SHOW, 1–7
/WARNINGS, 1–7
Macro-defined register symbols
list, 6–3
Macro definition, 4–1
default value, 4–2
ending, 5–36
labeling in, 4–7
Macro definition directive (.MACRO), 5–67
Macro deletion directive (.MDELETE), 5–71
MACRO directive, 1–2
.MACRO directive, 5–67
Macro exit directive (.MEXIT), 5–72
Macro expansion, 4–1
printing, 4–1
terminating, 5–72
Macro include directive (.INCLUDE), 5–53
Macro library directive (.LIBRARY), 5–59
Macro name, 2–9
Macros, 4–1
calling routines, 6–6
$CALL instruction sequence, 6–6
defining a routine, 6–5
ending routines, 6–2
for calling routines, 6–2
in epilogue sections, 6–5
in prologue sections, 6–4, 6–5
in source code, 6–7
MACRO–64 supplied, 6–1
MACRO–64 supplied library, 6–1
making multiple calls, 6–8
nested, 4–4, 5–68
passing numeric value to, 4–7
recursive, 5–68
redefining, 5–68
replacing predefined register symbols, B–2
supplied, 6–8 to 6–39
switching psects, 6–2, 6–5
syntax rules, 6–8
within routines, 6–1
with same name as Alpha AXP opcode, 5–68
with user-defined symbols, B–2
.MCALL directive, 5–70
.MDELETE directive, 5–71
Memory
accessing, B–4
Message display directive
.ERROR, 5–39
.PRINT, 5–82
Messages
errors, E–1 to E–20
Message warning display directive (.WARN), 5–102
.MEXIT directive, 5–72
Migration
alternatives to using VAX instruction-generating macros, B–2
avoiding misleading diagnostics, B–2
avoiding VAX instruction-generating macros, B–2

N

NAMES argument, 6–8
/NAMES qualifier, 1–5
.NARG directive, 5–74
.NCHR directive, 5–75
.NLIST directive, 5–76
NONSTANDARD argument, 6–13
.NOSHOW directive, 5–77, 5–94
Number
  See also Integer, Floating-point number, and Packed decimal string
  in source statement, 2–5
Number of arguments directive (.NARG), 5–74
Number of characters directive (.NCHR), 5–75
Numeric complement operator, 2–18
Numeric symbol
  with lexical string symbol, 3–3
Numeric symbols, 3–3

O

Object module
  identifying, 5–44
  naming, 5–100
  title, 5–100
/OBJECT qualifier, 1–5
Obtaining information lexical
  %TYPE, 3–19
Obtaining register number
  floating-point (%FREG), 3–11
  integer (%IREG), 3–13
.OCTA directive, 5–78, B–14
Octaword storage directive (.OCTA), 5–78
.ODD directive, 5–79
One's complement
  of expression, 2–18
Opcode
  common architecture, A–6 to A–11
  redefining, 5–68
  with the same name as a macro, 5–68
$OPDEF macro, 6–22
OpenVMS PALcode instruction summary, A–11 to A–15
Operand, 2–3
  instruction notation, A–4
Operand field, 2–1, 2–3
  using symbols in, 2–3
Operator, 2–3
  AND, 2–19
  arithmetic shift, 2–19
  binary, 2–18
  complement, 2–18
  exclusive OR, 2–19
  inclusive OR, 2–19
  radix control, 2–17
  unary, 2–17
Operator field, 2–1, 2–3
  using symbols in, 2–3
Optimization, B–13
  automatic code alignment, B–13
  automatic data alignment, B–13
  controlling, B–13
  default command-line qualifiers, B–13
  inlining, B–16
  loop unrolling, B–16
  peepholing, B–13
  scheduling, B–13
  viewing results, B–17
/OPTIMIZE qualifier, 1–5
OTS$DIV_I routine
  performing longword integer division, B–2
OTS$DIV_L routine
  performing quadword integer division, B–2

P

Packed decimal string, 5–80
.PACKED directive, 5–80
.PACKED macro, 6–26
Packed storage directive (.PACKED), 5–80
.PAGE directive, 5–81
Page ejection directive (.PAGE), 5–81
Period (.)
  current location counter, 2–20
Permanent symbol, 2–9
Permanent symbols, 2–7
Portability issues, D–1 to D–6
Positional argument, 4–3
Predefined symbols, 2–7
/PREPROCESSOR_ONLY qualifier, 1–6
.PRINT directive, 5–82
.PROCEDURE_DESCRIPTOR directive, 5–83, B–14
$PROCEDURE_DESCRIPTOR macro, 6–28
Programming hints, B–1
  example programs, B–1
Program section
  attributes, 5–84
  code, 6–2
  data, 6–2
  defining, 5–84
  directive
    .PSECT, 5–84
    .RESTORE_PSECT, 5–91
    .SAVE_PSECT, 5–92
  linkage, 6–2
  name, 5–87
  restoring context of, 5–91
  saving context of, 5–92
  saving local label, 5–92
Prologues
  entry and exit, B–11

Prologue sequence, B–7, B–11
ending, 6–18
.PSECT directive, 5–84
Pseudo-operations
decodable, A–6

Q

.QUAD directive, 5–88, B–14
Quadword storage directive (.QUAD), 5–88
Quoted string literal
with lexical string symbol, 3–3

R

Radix control operators, 2–17
Range extraction lexical
%EXTRACT, 3–10
Register name, 2–7, 2–9
Register symbols
definitions of, 6–3
macro-defined, 6–3
using in macros, B–2
Register usage
conventions, A–4
Relocatable expression, 2–15
Repeat block
argument substitution, 5–55
character substitution, 5–57
terminating repetition, 5–72
Repeat block directive (.REPEAT), 5–89
.REPEAT directive, 5–89
%REPEAT lexical, 3–16
Repeat range end directive (.ENDR), 5–37
.REPT directive, 5–89
$RESET_LP_LIST macro, 6–29
.RESTORE directive, 5–91
.RESTORE_PSECT directive, 5–91
$RETURN macro, 6–5, 6–30
Rls argument, 6–6, 6–10
$ROUTINE macro, 6–4, 6–31
qualifiers, 6–31 to 6–37
Routines, B–10
See Table B–1
bound-frame, B–11
defining type by macro, 6–4
null-frame, B–10
performance, B–11
register-frame, B–10
stack-frame, B–10
$RSA_END symbol, 6–4, 6–38
$RSA_OFFSET symbol, 6–3, 6–4, 6–38

S

.SAVE directive, 5–92
SAVED_REGS argument, 6–5
.SAVE_PSECT directive, 5–92
.SBTTL directive, 5–98
SCRATCH_REGS argument, 6–8, 6–12
SET_ARG_INFO argument, 6–11
Shift operator, 2–19
.SHOW directive, 5–94
/SHOW qualifier, 1–7
SIGNATURE_BLOCK argument, 6–13
Signed byte data directive (.SIGNED_BYTE), 5–96
Signed word storage directive (.SIGNED_WORD), 5–97
.SIGNED_BYTE directive, 5–96
.SIGNED_WORD directive, 5–97, B–14
Single-precision IEEE directive
.S_FLOATING, 5–93
$SIZE symbol, 6–3, 6–4, 6–38
Source statement
character set in, 2–4
format, 2–1
rules for coding, 2–2
$STACK_ARG_SIZE symbol, 6–4
STACK_RETURN_VALUE argument, 6–12
STANDARD_PROLOGUE argument, 6–4
Statements
character set, 2–4
comment, 2–4
continuation of, 2–2
field, 2–1
format, 2–1
label, 2–2
operand, 2–3
operator, 2–3
rules for coding, 2–2
types, 1–1
String arguments, 4–3
String editing lexical
%EDIT, 3–8
%STRING lexical, 3–17
String locating lexical
%LOCATE, 3–15
String repeating lexical
%REPEAT, 3–16
Strings
date/time lexical (%TIME), 3–18
determining length (%LENGTH), 3–14
editing lexical (%EDIT), 3–8
extracting elements (%ELEMENT), 3–9
extracting range (%EXTRACT), 3–10
locating lexical (%LOCATE), 3–15
obtaining information lexical (%TYPE), 3–19
obtaining value lexical (%STRING), 3–17
repeating lexical (%REPEAT), 3–16

Subconditional assembly block directive
.IF_FALSE, 5–49
.IF_TRUE, 5–49
.IF_TRUE_FALSE, 5–49
.SUBTITLE directive, 5–98
Subtitle listing control directive (.SUBTITLE), 5–98
Symbol attribute directive (.WEAK), 5–103
Symbols, 2–7
defined by $CALL, 6–4
defined by $ROUTINE, 6–3
defining register names, 2–8
deleting register names, 2–8
determining value of, 2–9
external, 5–41, 5–103
global, 2–10, 5–103
in operand field, 2–9
in operator field, 2–9
local, 2–10
macro name, 2–9
permanent, 2–7, 2–9
predefined, 2–7
register name, 2–7, 2–9
undefined, 5–28
user-defined, 2–8, 2–9
.S_FLOATING directive, 5–93, B–14

T

Tab stops
in source statement, 2–1
Temporary label, 2–3
user-defined, 2–10
Term in MACRO statement, 2–14
TIE argument, 6–12
%TIME lexical, 3–18
.TITLE directive, 5–100
Title listing control directive (.TITLE), 5–100
%TYPE lexical, 3–19
.T_FLOATING directive, 5–99, B–15

U

Unary operators, 2–17
Undefine register symbol directive (.UNDEFINE_REG), 5–101
.UNDEFINE_REG directive, 5–101
User-defined register symbols
in macros, B–2
User-defined symbols, 2–8, 2–9
User-defined temporary label, 2–10
range, 2–11
USES_VAX_ARGLIST argument, 6–13

V

Value conversion lexical
%INTEGER, 3–12
Value obtaining lexical
%STRING, 3–17
Values
converting (%INTEGER), 3–12
default, 4–2
VAX MACRO features
not in MACRO–64, D–4
arguments to .DISABLE directive, D–5
arguments to .ENABLE directive, D–5
count specifiers on data-storage directives, D–4
.CROSS directive, D–5
.DEBUG directive, D–5
.DEFAULT directive, D–5
.END directive differences, D–5
.ENDM directive differences, D–6
.ENDR directive differences, D–6
.ENTRY directive, D–5
EXE/NOEXE restriction, D–4
.GLOBAL directive, D–5
.H_FLOATING directive, D–5
.LINK directive, D–5
.MASK directive, D–5
.NOCROSS directive, D–5
.NTYPE directive, D–5
.OPDEF directive, D–5
overlapping and replacement initializers, D–4
PAGE argument to .ALIGN directive, D–4
.PSECT directive differences, D–5
radix unary operators, D–4
.REFn directives, D–6
register mask operator, D–4
.RESTORE directive differences, D–6
setting current location counter, D–4
.TRANSFER directive, D–6

W

.WARN directive, 5–102
/WARNINGS qualifier, 1–7
.WEAK directive, 5–103
.WORD directive, 5–104, B–14
Word storage directive (.WORD), 5–104