

# **HP Assembler Reference Manual**

## **HP 9000 Computers**

**9th Edition**



**92432-90012**

**June 1998**

Printed in: U.S.A.

© Copyright 1998 Hewlett-Packard Company. All rights reserved.

---

## Legal Notices

The information contained in this document is subject to change without notice.

*Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.*

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains information which is protected by copyright. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

---

## Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Rights for non-DOD U.S. Government Departments and Agencies are set forth in FAR 52.227-19(c)(1,2).

*HEWLETT-PACKARD COMPANY  
3000 Hanover Street  
Palo Alto, California 94304  
U.S.A.*

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

---

# Contents

<b>Preface</b> .....	<b>11</b>
Printing History .....	11
Audience .....	12
Related Documentation .....	12
Typographical Conventions .....	13
In This Manual .....	13
Summary of Technical Changes for HP-UX 11.0 .....	14
<b>1. Introduction to PA-RISC Assembly Language</b>	
Assembler Features .....	15
Summary of Changes for PA-RISC 2.0 .....	17
Summary of Changes for PA-RISC 2.0W (Wide Mode, 64-bit) .....	17
<b>2. Program Structure</b>	
Symbols and Constants .....	21
Registers and Register Mnemonics .....	23
Expressions .....	29
Parenthesized Subexpressions .....	34
Operands and Completers .....	35
Macro Processing .....	37
Defining New Instructions With Macros .....	37
<b>3. HP-UX Architecture Conventions</b>	
Spaces .....	39
Subspaces .....	42
Attributes .....	42

---

## Contents

Directives . . . . .	43
Sections in 64-bit Mode . . . . .	44
Location Counters . . . . .	46
Compiler Conventions . . . . .	47
Shared Libraries. . . . .	51
Assembly Listing . . . . .	51
<b>4. Assembler Directives and Pseudo-Operations</b>	
Introduction . . . . .	53
.ALIGN Directive . . . . .	57
Syntax . . . . .	57
Parameters . . . . .	57
Example. . . . .	57
.ALLOW Directive . . . . .	58
Syntax . . . . .	58
Parameters . . . . .	58
Discussion . . . . .	58
Example. . . . .	59
.BLOCK and .BLOCKZ Pseudo-Operations . . . . .	60
Syntax . . . . .	60
Parameters . . . . .	60
Discussion . . . . .	60
Example. . . . .	61
.BYTE Pseudo-Operation. . . . .	62
Syntax . . . . .	62
Parameters . . . . .	62
Discussion . . . . .	62
Example. . . . .	62

---

## Contents

.CALL Directive . . . . .	63
Syntax . . . . .	63
Parameters . . . . .	63
Example . . . . .	64
.CALLINFO Directive . . . . .	67
Syntax . . . . .	67
Parameters . . . . .	67
Discussion . . . . .	71
Example . . . . .	72
.COMM Directive . . . . .	74
Syntax . . . . .	74
Parameters . . . . .	74
Discussion . . . . .	74
Example . . . . .	74
.COPYRIGHT Directive . . . . .	75
Syntax . . . . .	75
Parameters . . . . .	75
Discussion . . . . .	75
Example . . . . .	76
.DOUBLE Pseudo-Operation . . . . .	77
Syntax . . . . .	77
Parameters . . . . .	77
Example . . . . .	77
.DWORD Pseudo-Operation . . . . .	78
Syntax . . . . .	78
Parameters . . . . .	78
Discussion . . . . .	78
Example . . . . .	78
.END Directive . . . . .	79

---

## Contents

Syntax .....	79
Discussion .....	79
Example .....	79
.ENDM Directive .....	80
Syntax .....	80
Example .....	80
.ENTER and .LEAVE Pseudo-Operations .....	81
Syntax .....	81
Discussion .....	81
Example .....	82
.ENTRY and .EXIT Directives .....	83
Syntax .....	83
Discussion .....	83
Example .....	83
.EQU Directive .....	84
Syntax .....	84
Parameters .....	84
Example .....	84
.EXPORT Directive .....	85
Syntax .....	85
Parameters .....	85
Discussion .....	87
Example .....	87
.FLOAT Pseudo-Operation .....	88
Syntax .....	88
Parameters .....	88
Example .....	88
.HALF Pseudo-Operation .....	89
Syntax .....	89

---

## Contents

Parameters .....	89
Discussion .....	89
Example .....	89
.IMPORT Directive .....	90
Syntax .....	90
Parameters .....	90
Discussion .....	91
Example .....	91
.LABEL Directive .....	92
Syntax .....	92
Parameters .....	92
Example .....	92
.LEVEL Directive .....	93
Syntax .....	93
Parameters .....	93
Discussion .....	93
.LISTOFF and .LISTON Directives .....	95
Syntax .....	95
Example .....	95
.LOCCT Directive .....	97
Syntax .....	97
Parameters .....	97
Example .....	97
.MACRO Directive .....	98
Syntax .....	98
Parameters .....	98
Discussion .....	98
Examples .....	99
.ORIGIN Directive .....	101

---

## Contents

Syntax .....	101
Parameters .....	101
Discussion .....	101
Example .....	101
.PROC and .PROCEND Directives .....	102
Syntax .....	102
Discussion .....	102
Example .....	103
.REG Directive .....	104
Syntax .....	104
Parameters .....	104
Example .....	104
.SHLIB_VERSION Directive .....	105
Syntax .....	105
Parameters .....	105
Example .....	105
.SPACE Directive .....	106
Syntax .....	106
Parameters .....	106
Discussion .....	107
Example .....	107
.SPNUM Pseudo-Operation .....	108
Syntax .....	108
Parameters .....	108
Example .....	108
.STRING and .STRINGZ Pseudo-Operations .....	109
Syntax .....	109
Parameters .....	109
Discussion .....	110



---

## Contents

Examples . . . . .	110
.SUBSPA Directive . . . . .	111
Syntax . . . . .	111
Parameters . . . . .	111
Discussion . . . . .	113
Example . . . . .	113
.VERSION Directive . . . . .	114
Syntax . . . . .	114
Parameters . . . . .	114
Discussion . . . . .	114
Example . . . . .	114
.WORD Pseudo-Operation . . . . .	115
Syntax . . . . .	115
Parameters . . . . .	115
Discussion . . . . .	115
Example . . . . .	115
Programming Aids . . . . .	116
<b>5. Pseudo-Instruction Set</b>	
<b>6. Assembling Your Program</b>	
Invoking the Assembler . . . . .	123
Using the as Command . . . . .	124
Syntax . . . . .	124
Parameters . . . . .	124
Using the cc Command . . . . .	127
Passing Arguments to the Assembler . . . . .	127
cpp Preprocessor . . . . .	128

---

## Contents

<b>7. Programming Examples</b>	
1. Binary Search for Highest Bit Position. . . . .	130
2. Copying a String. . . . .	132
3. Dividing a Double-Word Dividend. . . . .	134
4. Demonstrating the Procedure Calling Convention . . . . .	136
C Program Listing. . . . .	136
Assembly Program Listing. . . . .	137
5. Output of the cc -S Command . . . . .	138
C Program Listing. . . . .	138
Assembly Program Listing From the C Compiler. . . . .	138
<b>8. Diagnostic Messages</b>	
Warning Messages . . . . .	142
Error Messages. . . . .	154
Panic Messages. . . . .	176
User Warning Messages. . . . .	178
Limit Error Messages . . . . .	183
Branching Error Messages . . . . .	187
<b>Index . . . . .</b>	<b>195</b>

---

## Preface

This manual describes the use of the Precision Architecture RISC (PA-RISC) Assembler on HP 9000 computers.

You need to be familiar with the machine instructions to use the Assembler. For a complete description of the machine instruction set, refer to *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and *PA-RISC 2.0 Architecture*.

Note that, throughout this manual, there are references to PA-RISC 1.0, 1.1, and 2.0. Each version of the architecture is a superset of the preceding version.

Any program written for PA-RISC 1.0 machines will execute on PA-RISC 1.1 and 2.0 machines, but programs using instructions unique to PA-RISC 1.1 will not execute on PA-RISC 1.0 machines. Any program written for PA-RISC 1.1 machines will execute on PA-RISC 2.0 machines, but programs using features unique to PA-RISC 2.0 will not execute on PA-RISC 1.1 or 1.0 machines.

## Printing History

New editions are complete revisions of the manual. Technical addendums or release notes may be released as supplements.

The software version is the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

<b>Edition</b>	<b>Date</b>	<b>Software Version</b>
First Edition	November 1986	
Update 1	March 1987	
Update 1 Incorporated	May 1987	

<b>Edition</b>	<b>Date</b>	<b>Software Version</b>
Second Edition	January 1988	92432-03A.00.03
Third Edition	November 1988	92432-03A.00.04
Fourth Edition	January 1991	92432-03A.08.06
Fifth Edition	January 1995	92432-03A.10.00
Sixth Edition	June 1996	92432-03A.10.20
Seventh Edition	May 1997	92432-03A.10.30
Eighth Edition	November 1997	92453-03A.11.00
Ninth Edition	June 1998	92453-03A.11.00

You may send any suggestions for improvements in this manual to:

Languages Information Engineering Manager  
Hewlett-Packard Company  
Mailstop 42UD  
11000 Wolfe Road  
Cupertino CA 95014-9804

Electronic Mail: editor@cup.hp.com

## **Audience**

This manual assumes that you are an experienced assembly language programmer. In addition, you should have detailed understanding of the PA-RISC and hardware features, and a working knowledge of the HP-UX operating system, program structures, procedure calling conventions, and stack unwind procedures.

## **Related Documentation**

For more information on HP-UX programming, refer to the following documents:

- *PA-RISC 2.0 Architecture* by Gerry Kane (Prentice-Hall, ISBN 0-13-182734-0)

- *HP-UX Linker and Libraries Online User Guide*, (ld +help)
- *64-bit Runtime Architecture for PA-RISC 2.0*. URL: <http://www.software.hp.com/STK/>
- *ELF 64 Object File Format*. URL: <http://www.software.hp.com/STK/>

## Typographical Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

<code>computer font</code>	Computer font indicates commands, keywords, options, literals, source code, system output, and path names. In syntax formats, computer font indicates commands, keywords, and punctuation that you must enter exactly as shown.
<b>bold face text</b>	In examples, bold face text represents user input.
<i>italic type</i>	In syntax formats, words or characters in italics represent values that you must supply. Italics are also used for book titles and for emphasis.
[    ]	In syntax formats, square brackets enclose optional items.
{    }	In syntax formats, braces enclose a list from which you must choose an item.
...	In syntax formats, a horizontal ellipsis indicates that you can repeat the preceding item one or more times.
<i>name</i> (N)	An italicized word followed by a number in parentheses indicates an entry in <i>HP-UX Reference</i> . For example, <i>cc</i> (1) refers to the <i>cc</i> entry in Section 1 of <i>HP-UX Reference</i> .

## In This Manual

The manual is organized as follows:

Chapter 1	Introduces the Assembler for HP 9000 computers.
Chapter 2	Explains assembly language program structure.

Chapter 3	Explains programming in Assembler for HP-UX.
Chapter 4	Describes the PA-RISC Assembler directives and pseudo-operations.
Chapter 5	Summarizes the pseudo-instructions for the PA-RISC machine instructions.
Chapter 6	Describes the assembly ( <code>as</code> ) command and the ways to invoke the PA-RISC Assembler under the HP-UX operating system.
Chapter 7	Contains several sample assembly language programs.
Chapter 8	Lists the diagnostic messages that the PA-RISC Assembler can generate.

## Summary of Technical Changes for HP-UX 11.0

The following features have changed for HP-UX 11.00 to support PA-RISC 2.0W (wide), 64-bit mode. These changes are explained in detail in the appropriate locations in this manual.

- In 64-bit mode, the linkage pointer register is `%r27`. See Table 2-11, “Available Field Selectors,” on page 31.
- In 64-bit mode, the Executable and Linking Format (ELF) uses segments and sections rather than spaces and subspaces. See “Sections in 64-bit Mode” on page 44.
- The Assembler ignores the `.CALL` directive. This means that your program must ensure that the caller and called procedure agree on argument locations. See “.CALL Directive” on page 63.
- The `.CALLINFO` directive parameters include updates to support 64-bit mode.
- You can specify 2.0W with the `.LEVEL` directive to tell the the Assembler to generate 64-bit object code. For details, see “.LEVEL Directive” on page 93.
- New and changed Assembler error messages. For details, see Chapter 8, “Diagnostic Messages,” on page 141.

---

# 1

## Introduction to PA-RISC Assembly Language

The HP 9000 Assembly Language represents machine language instructions symbolically, and permits declaration of addresses symbolically as well. The Assembler's function is to translate an assembly language program, stored in a *source file*, into machine language. The result of this translation resides in a *relocatable object file*. The object file is relocatable because it can still be combined with other relocatable object files and libraries. Therefore, it is necessary to relocate any addresses that the Assembler chooses for the symbols in the source program.

This process of combining object files and libraries is performed by the linker, `ld`. The linker's task is to transform one or more relocatable object files into an executable *program file*. Every program must be linked before it can be executed, even if the source file is complete within itself and does not need to be combined with other files.

---

### Assembler Features

The Assembler provides a number of features to make assembly language programming convenient. These features include:

- **Mnemonic Instructions.** Each machine instruction is represented by a mnemonic operation code, which is easier to remember than the binary machine language operation code. The operation code, together with operands, directs the Assembler to output a binary machine instruction to the object file.
- **Symbolic Addresses.** You can select a symbol to refer to the address of a location in virtual memory. The address is often referred to as the *value* of the symbol, which should not be confused with the value of the memory locations at that address. These symbols are called *relocatable symbols* because the actual addresses represented by such symbols are subject to relocation by the linker.
- **Symbolic Constants.** A symbol can also be selected to stand for an integer constant. These symbols are called *absolute symbols* because the values of such symbols are not relocatable.

## Assembler Features

- **Expressions.** Arithmetic expressions can be formed from symbolic addresses and constants, integer constants, and arithmetic operators. Expressions involving only symbolic and integer constants, or the difference between two relocatable symbols, defined in the current module, are called *absolute expressions*. They can be used wherever an integer constant can be used. Expressions involving the sum or difference of a symbolic address and an absolute expression are called *relocatable expressions* or *address expressions*. The constant part of an expression, the part that does not refer to relocatable expressions, can use parenthesized subexpressions to alter operator precedence.
- **Storage Allocation.** In addition to encoding machine language instructions symbolically, storage may be initialized to constant values or simply reserved. Symbolic addresses and *labels* can be associated with these memory locations.
- **Symbol Scope.** When two or more object files are to be combined by the linker, certain symbolic addresses can be defined in one module and used in another. Such symbols must be *exported* from the defining module and *imported* into the using module. In the defining module, the symbol has *universal* scope, while in the using module, the symbol is *unsatisfied*. Other symbols declared in the source program that are not exported have *local* scope.
- **Subspaces and Location Counters.** You can organize code and data into separate subspaces, and into separate location counters within each subspace. The programmer can move among the subspaces and location counters, while the Assembler changes the code and data into the correct order. In 64-bit mode, however, the Executable and Linking Format (ELF) uses segments and sections rather than spaces and subspaces.
- **Macro Processing.** A *macro* is a user-defined word, which is replaced by a sequence of instructions. Including a macro in a source program causes the sequence of instructions to be inserted into the program wherever the macro appears.



## Summary of Changes for PA-RISC 2.0

The following features have changed in PA-RISC 2.0 architecture. These changes are explained in more detail in the appropriate locations in this manual.

- A new `.DWORD` directive reserves 64 bits (a double word) of storage and initializes it to the given value.
  - A `.LEVEL 2.0` directive should be used as the first directive in the source file to assemble it for a PA-RISC 2.0 system.
  - New `+DA2.0` option
  - New and changed Assembler error messages
- 

## Summary of Changes for PA-RISC 2.0W (Wide Mode, 64-bit)

The Assembler for PA-RISC 2.0W, the 64-bit version of PA-RISC 2.0, maintains the same source syntax as that of PA1.x and PA2.0 32-bit mode versions. However, PA2.0W features differ in the features listed below.

- To assemble a source file for a PA-RISC 64-bit system, use a `.LEVEL 2.0W` directive as the first directive in the source file. See “`.LEVEL Directive`” on page 93.
- The Assembler generates an Executable and Linking Format (ELF) object file format with PA-RISC 2.0W. Refer to the *ELF 64 Object File Format*, URL: <http://www.software.hp.com/STK/> for details on ELF format.
- PA-RISC 2.0W supports a flat virtual address space of  $2^{**}64$  bytes, and therefore does *not* support use of space registers. Use the following syntax when memory operations are used:

ex: `LDD disp(b), tgt`

You can explicitly use space registers, however, the Assembler issues a warning if it is other than `sr0`.

- Some of the completers on `ADDB` and `ADDIB` instructions are not valid for PA2.0W. In addition, new completers are available.

For example: `ZNV`, `SV`, and `OD` are not valid whereas `*=`, `*<`, and `*<=` are additional completers.

Please refer to the *PA-RISC 2.0 Architecture* guide for details.

- The displacement on both general load/store and floating load/store instructions can be up to 16 bits. For example,

```
ex: FLDD disp(b),tgt      ; displacement can be up to 16 bits.
```

Please refer to *PA-RISC 2.0 Architecture* for details.

- You must change any `.WORD` directives that are initialized with a code symbol or data symbol to `.DWORD`.
- You can not use space identification operations such as `MTSP` and `LDSID` used for dealing with space registers in user level code. Currently, the Assembler does *not* give any warning.
- The procedure calling conventions are different in the HP-UX PA-RISC 2.0 64-bit architecture. In PA 2.0W, you can pass the first eight parameters in registers (`arg0-arg7`). In earlier versions (PA1.0 and PA1.1) and on PA-RISC 2.0, you can only pass the first four parameters in registers(`arg0-arg3`). For more information, please refer to the *64-bit Runtime Architecture for PA-RISC 2.0*, at URL: <http://www.software.hp.com/STK/>.

---

## Program Structure

An assembly language program is a sequence of *statements*. There are three classes of statements:

- Instructions
- Pseudo-operations
- Directives

Instructions represent a single machine instruction in symbolic form. Pseudo-operations cause the Assembler to initialize or reserve one or more words of storage for data, rather than machine instructions. Directives communicate information about the program to the Assembler, but do not generally cause the Assembler to output any machine instructions.

An assembly statement contains four *fields*:

- Label
- Opcode
- Operands
- Comments

Each of these fields is optional. However the operands field cannot appear without an opcode field. The *label* field is used to associate a symbolic address with an instruction or data location, or to define a symbolic constant using the `.EQU`, `.REG`, or `.MACRO` directives. This field is optional for all but a few statement types; if present, the label must begin in column one of a source program line. If a label appears on a line by itself, or with a comment only, the label is associated with the next address within the same subspace and location counter.

When the label field begins with the pound sign (#) character, it is not treated as a label. If # is followed by white space and an integer, the Assembler's line number counter, used when reporting errors, is reset to the value of the integer. Otherwise, the line beginning with # is ignored. This feature is for the use of the C language preprocessor `cpp`.

The *opcode* field contains either a mnemonic machine instruction, a pseudo-operation code, or the name of an Assembler directive. It must be separated from the label field by a blank or tab. For certain machine instructions, the opcode field can also contain *completers*, separated from the instruction mnemonic by commas.

## Program Structure

The machine instruction mnemonics and completers are described in the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and *PA-RISC 2.0 Architecture*.

The *operands* field follows the opcode field, separated by a blank or tab. Operands are separated by commas. The meaning of the operands depends on the specific statement type, determined by the opcode.

The number of operands permitted or required depends upon the specific instruction.

The *comments* field is introduced with a semicolon, and causes the Assembler to ignore the remainder of the source line. A comment can appear on a line by itself.

The following listing contains several assembly language statements. The headings identify the four fields.

<b>Label</b>	<b>Opcode</b>	<b>Operands</b>	<b>Comments</b>
JAN	.EQU	1	;declares a symbolic constant
SUM	.WORD	0	;reserve a word and set to zero
LOOP	LDW	4(%r1),%r2	
	ADD	%r2,%r3,%r4	
	STW	%r4,SUM- $\$$ global $\$$ (%dp)	
	BL	LOOP,%r0	

Statements are normally written on separate lines. It is sometimes useful, especially when using a macro preprocessor, to be able to write several statements on one line. This can be done by separating the statements with the “!” character. When this feature is used, a label can be placed only on the first statement of the line, and a comment can only follow the last statement on the line. The `.LABEL` directive can override this condition by providing a means for declaring a label within a multi-statement line.

## Symbols and Constants

Both addresses and constants can be represented symbolically. Labels represent a symbolic address except when the label is on an `.EQU`, `.REG`, or `.MACRO` directive. If the label is on an `.EQU` or `.REG` directive, the label represents a symbolic constant. If the label is the `.MACRO` directive, the label represents a macro name.

Symbols are composed of uppercase and lowercase letters (A-Z and a-z), decimal digits (0-9), dollar signs (\$), periods (.), ampersands (&), pound signs (#), and underscores (\_). A symbol can begin with a letter, digit, underscore, or dollar sign. If a symbol begins with a digit it must contain a non-digit character. (The predefined register symbols begin with a percent sign (%).)

The Assembler considers uppercase and lowercase letters in symbols to be distinct. The mnemonics for operation codes, directives, and pseudo-operations can be written in either case. There is no explicit limit on the length of a symbol. The following are examples of legal symbols:

```
$START$  _start  PROGRAM  M$3      $global$  
$$mulI   main    P_WRITE  loop1    1st_time
```

The following are examples of illegal symbols:

```
LOOP|1    Contains an illegal character  
&st_time Begins with &  
123      Does not contain a nondigit
```

Integer constants can be written in decimal, octal, or hexadecimal notation, as in the C language. “Integer Constants” on page 22 lists the ranges of these integer constants.

**Table 2-1 Integer Constants**

	<b>Signed</b>	<b>Unsigned</b>
<b>Decimal</b>	-2147483648 <b>through</b> 2147483647	0 <b>through</b> 4294967295
<b>Octal</b>	020000000000 <b>through</b> 017777777777	0 <b>through</b> 037777777777
<b>Hexadecimal</b>	0x80000000 <b>through</b> 0x7FFFFFFF	0 <b>through</b> 0xFFFFFFFF

The period (.) is a special symbol reserved to denote the current offset of the location counter. It is useful in address expressions to refer to a location relative to the current instruction or data word. This symbol is considered relocatable, and can be used anywhere a relocatable symbol can be used, with the exception of the label field.

The period cannot be used in an expression involving another label, such as *sym+.*, *sym-.*, *+.sym*, or *-.sym*. It can be used in an expression that has only a constant, such as *+.8* or *-.8*.

## Registers and Register Mnemonics

PA-RISC processors have four sets of registers:

- General
- Floating-point
- Space
- Control

Data is loaded from memory into general or floating-point registers and stored into memory from general or floating-point registers. Arithmetic and logical operations are performed on the contents of the general registers. On PA-RISC 1.0 or 1.1 each general register is 32 bits wide. On PA-RISC 2.0 each general register is 64 bits wide. On PA-RISC 2.0W (true 64-bit environment) each general register is 64 bits wide.

There are 32 general registers, denoted as %r0 through %r31. General register %r0 is special because “writes” into it are ignored, and it always reads as zero. The remaining general registers can be used normally, with the caution that %r1 is the implicit target register for the ADDIL instruction, %r31 is the implicit link register for the BLE instruction, and for PA-RISC 2.0 only, %r2 is the implicit link register for the BLVE instruction. Certain general registers also have predefined conventional uses. Refer to “Register Procedure Calling Conventions” on page 28. You can find detailed information on both 32-bit and 64-bit runtime architecture under the topic PA-RISC Architecture at <http://www.software.hp.com/STK/>.

PA-RISC 1.0 machines have 16 floating-point registers; PA-RISC 1.1, 2.0, and 2.0W (true 64-bit environment) machines have 32 floating-point registers. Each register is capable of holding either a single- or double-precision floating-point number in IEEE format. These registers are denoted %fr0 through %fr15 for PA-RISC 1.0 and %fr0 through %fr31 for PA-RISC 1.1, 2.0, and 2.0W.

Registers %fr1, %fr2, and %fr3 are exception registers and are not available to the programmer. Floating-point register %fr0 contains a permanent floating-point zero when used in an arithmetic operation; when written or read with floating-point loads or stores, the floating-point status register is actually accessed.

In addition, on PA-RISC 1.1, 2.0, and 2.0W the left and right halves of the floating-point registers can be accessed as separate single-precision registers by using an L or R suffix.

For example, `%fr8R` accesses the right-most 32 bits of `%fr8` as a single-precision number.

The L or R suffixes can only be used on the predefined floating-point registers in the form `%frnn`, where *nn* is the register number. It is not legal to use L or R with an integer value. For example, `%fr8R` is legal; `8R` is not legal.

The space registers form the basis of the virtual memory system. Each of the eight space registers can hold a 16- or 32-bit space identifier, depending on the hardware model. The space registers are denoted as `%sr0` through `%sr7`. Space register `%sr0` is set implicitly by the `BLE` instruction, and space registers `%sr5` through `%sr7` cannot be modified except by code running at the most privileged level.

The control registers contain system-state information. There are 25 control registers, denoted as `%cr0` and `%cr8` through `%cr31`. Of these registers, only `%cr11` (`%sar`), the shift amount register, and `%cr16` (`%itmt`), the interval timer, are normally accessible to the user-level programmer. The other registers are accessed only by code running at the most privileged level.

Register operands are denoted by register-typed constants because the Assembler needs to be able to differentiate between general registers, space registers, floating point registers, and ordinary integer constants.

To make assembly code more readable, you can use the `.REG` directive to declare a symbolic name as an alias for a predefined register. The predefined registers have a register type associated with them. The Assembler enforces register type checking and issues a warning message if the wrong kind of register is used within an operand. A warning is also issued when an integer constant or absolute expression is found where a register is expected. You must use the `.REG` directive to define symbolic register names. If a symbolic name defined in an `.EQU` directive is used where a register symbol is expected, the Assembler issues a warning message, because it considers an `.EQU` defined symbol to be a simple integer constant.

---

NOTE

If an absolute expression is used instead of a register or register-typed symbol name, the Assembler issues warning message number 41.



This warning can be suppressed with the `-w41` command-line option. Future versions of the Assembler may not always allow an absolute expression where a register is expected.

The following example demonstrates the correct usage of the `.REG` directive:

```
tblptr    .REG    %r20  
aka_tbl   .REG    tblptr
```

Predefined registers are shown in the following tables. All of the mnemonics begin with the `%` character, so they do not conflict with any programmer-defined symbols.

**Table 2-2**

**General Registers**

%r0	%r8	%r16	%r24
%r1	%r9	%r17	%r25
%r2	%r10	%r18	%r26
%r3	%r11	%r19	%r27
%r4	%r12	%r20	%r28
%r5	%r13	%r21	%r29
%r6	%r14	%r22	%r30
%r7	%r15	%r23	%r31

**Table 2-3 Single-Precision Floating-Point Registers**

%fr0L	%fr8L	%fr16L	%fr24L
%fr1L	%fr9L	%fr17L	%fr25L
%fr2L	%fr10L	%fr18L	%fr26L
%fr3L	%fr11L	%fr19L	%fr27L
%fr4L	%fr12L	%fr20L	%fr28L
%fr5L	%fr13L	%fr21L	%fr29L
%fr6L	%fr14L	%fr22L	%fr30L
%fr7L	%fr15L	%fr23L	%fr31L
%fr0R	%fr8R	%fr16R	%fr24R
%fr1R	%fr9R	%fr17R	%fr25R
%fr2R	%fr10R	%fr18R	%fr26R
%fr3R	%fr11R	%fr19R	%fr27R
%fr4R	%fr12R	%fr20R	%fr28R
%fr5R	%fr13R	%fr21R	%fr29R
%fr6R	%fr14R	%fr22R	%fr30R
%fr7R	%fr15R	%fr23R	%fr31R
<p>Accessing the right half of floating-point registers separately is possible only on PA-RISC 1.1 or later architectures.</p> <p>Registers %fr16L through %fr31L and %fr16R through %fr31R are available only on PA-RISC 1.1 or later architectures.</p>			

**Table 2-4 Double-Precision Floating-Point Registers**

%fr0	%fr8	%fr16	%fr24
%fr1	%fr9	%fr17	%fr25
%fr2	%fr10	%fr18	%fr26
%fr3	%fr11	%fr19	%fr27
%fr4	%fr12	%fr20	%fr28
%fr5	%fr13	%fr21	%fr29
%fr6	%fr14	%fr22	%fr30
%fr7	%fr15	%fr23	%fr31
Registers %fr16 through %fr31 are available only on PA-RISC 1.1 or later architectures.			

**Table 2-5 Space Registers**

%sr0	%sr2	%sr4	%sr6
%sr1	%sr3	%sr5	%sr7

**Table 2-6 Control Registers**

Registers	Synonyms	Registers	Synonyms
%cr0	%rctr	%cr20	%isr
%cr8	%pidr1	%cr21	%ior
%cr9	%pidr2	%cr22	%ipsw
%cr10	%ccr	%cr23	%eirr
%cr11	%sar	%cr24	%tr0 %ppda
%cr12	%pidr3	%cr25	%tr1 %hta
%cr13	%pidr4	%cr26	%tr2
%cr14	%iva	%cr27	%tr3
%cr15	%eiem	%cr28	%tr4
%cr16	%itmr	%cr29	%tr5
%cr17	%pcsq	%cr30	%tr6
%cr18	%pcoq	%cr31	%tr7
%cr19	%iir		

Some additional predefined register mnemonics are provided in “Register Procedure Calling Conventions” on page 28 to match the standard procedure-calling convention. This is discussed briefly in “HP-UX Architecture Conventions” on page 39. You can find detailed information on both 32-bit and 64-bit calling conventions under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>.

**Table 2-7 Register Procedure Calling Conventions**

<b>Register</b>	<b>Synonyms</b>	<b>Description</b>
%fr4	%farg0 %fret	Floating argument, return value
%fr5	%farg1	Second floating argument
%fr6	%farg2	Third floating argument
%fr7	%farg3	Fourth floating argument
%r2	%rp	Return link
%r19	%t4	Fourth temporary register
%r20	%t3	Third temporary register
%r21	%t2	Second temporary register
%r22	%t1	First temporary register
%r23	%arg3	Argument word 3
%r24	%arg2	Argument word 2
%r25	%arg1	Argument word 1
%r26	%arg0	Argument word 0
%r27	%dp	Data pointer
%r28	%ret0	Return value
%r29	%ret1 %sl	Return value, static link
%r30	%sp	Stack pointer
%r31	%mrp	Millicode return link
%srl	%sret %sarg	Return value, argument

In addition, there is a special register mnemonic defined as %previous\_sp, that allows access to the previous value of the stack pointer.

%previous\_sp must be used in the position of a base register; it can be used only between .ENTER and .LEAVE pseudo-operations. %previous\_sp is the same as %sp unless the current .PROC has a large

frame (that is, `.CALLINFO` specified `FRAME > 8191`) or `.CALLINFO` specified `.ALLOCA_FRAME`. In those two cases, `%previous_sp` is the same as `%r3`, and `%r3` is set up by the `.ENTER` pseudo-operation.

---

## Expressions

Arithmetic expressions are often valuable in writing assembly code. The Assembler allows expressions involving integer constants, symbolic constants, and symbolic addresses. These terms can be combined with the standard arithmetic operators shown in “Standard Arithmetic Operators” on page 29 or with bit-wise operators shown in “Bit-Wise Operators” on page 29.

**Table 2-8**

**Standard Arithmetic Operators**

Operator	Operation
+	Integer addition
-	Integer subtraction
*	Integer multiplication
/	Integer division (result is truncated)

The multiplication and division operators have *precedence* over addition and subtraction. That is, multiplications and divisions are performed first from left to right, then additions and subtractions are performed from left to right. Therefore, the expression `2+3*4` evaluates to 14.

**Table 2-9**

**Bit-Wise Operators**

Operator	Operation
	Logical OR
&	Logical AND
~	Unary logical complement (tilde)

Expressions produce either an absolute or a relocatable result. Any operation involving only absolute terms yields an absolute result. Relocatable terms are allowed only for the + and - operators. The legal combinations involving relocatable terms are shown in “Legal Combinations For Relocatable Terms” on page 30.

**Table 2-10**      **Legal Combinations For Relocatable Terms**

Operation	Result
Absolute + Relocatable	Relocatable
Relocatable + Absolute	Relocatable
Relocatable - Absolute	Relocatable
Relocatable - Relocatable (defined locally)	Absolute

For more information on the term *relocatable*, refer to “Assembler Features” on page 15.

---

**NOTE**

---

The combination “relocatable-relocatable+relocatable” is not permitted.

For example, assume the symbols MONTH and YEAR are relocatable, and JANUARY and FEBRUARY are absolute. The expressions MONTH+JANUARY and MONTH+FEBRUARY-4 are relocatable, while the expressions YEAR-MONTH and FEBRUARY-4 are absolute. The expression MONTH+JANUARY\*4 is also legal and produces a relocatable result, because JANUARY\*4 is evaluated first, producing an absolute intermediate result that is added to MONTH. The expression MONTH+YEAR is illegal, because the sum of two relocatable terms is not permitted.

Because all instructions are a single word in length, it is not possible to form a complete 32-bit address in a single instruction. Therefore, it is likely that the Assembler (or linker) may not be able to insert the final address of a symbol into the instruction as desired. For example, to load the contents of a word into a register, the following instruction could be used:

```
LDW            START, %r2
```

Because LDW provides only 14 bits for the address of START, the Assembler or linker prints an error message if the address of START requires more than 14 bits. There are two instructions, LDIL and ADDIL, whose function is to form the left-most 21 bits of a 32-bit address. The succeeding instruction, by using the target of the LDIL or ADDIL as a

base register, needs only 11 bits for the remainder of the address. The Assembler provides special operators, called *field selectors*, that extract the appropriate bits from the result of an expression. With the field selectors L' and R', the previous example can be recoded as follows:

```
LDIL      L'START,%r1          ;put left part into r1
LDW       R'START(%r1),%r2     ;add r1 and right part
```

The field selectors are always applied to the final result of the expression. They cannot be used in the interior of an expression. “Available Field Selectors” on page 31 shows all the available field selectors and their meanings.

**Table 2-11**

**Available Field Selectors**

<b>Field Selector</b>	<b>Meaning</b>
F' or F%	Full 32 bits (default).
L' or L%	Right-justified, high-order 21 bits.
R' or R%	Low-order 11 bits.
LS' or LS%	High-order 21 bits after rounding to nearest multiple of 2048.
RS' or RS%	Low-order 11 bits, sign extended.
LD' or LD%	Right-justified, high-order 21 bits after rounding to next multiple of 2048.
RD' or RD%	Low-order 11 bits, with negative sign.
LR' or LR%	L% value with constant rounded to nearest multiple of 8192.
RR' or RR%	R% value with constant rounded to nearest multiple of 8192, plus the difference of the constant and the rounded constant.
T' or T%	F% value offset of data linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.

Program Structure  
Expressions

<b>Field Selector</b>	<b>Meaning</b>
LT' or LT%	LR% value offset of data linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.
RT' or RT%	RR% value offset of data linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.
Q' or Q%	F% value offset of procedure linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.
LRQ' or LRQ%	LR% value offset of procedure linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.
RRQ' or RRQ%	RR% value offset of procedure linkage table slots from linkage table pointer. In 32-bit mode, the linkage table pointer is %r19. In 64-bit mode, the linkage table pointer is %r27.
P' or P%	Data procedure label (plabel) constructor.
LP' or LP%	Code procedure label (plabel) constructor used in LDIL instruction.
RP' or RP%	Code procedure label (plabel) constructor used in LDO instruction.
N' or N%	A null field selector, which is applied to an LDO instruction to allow a three-instruction sequence.
NL' or NL%	Right-justified, high-order 21 bits; allows a three-instruction sequence.



Field Selector	Meaning
NLD' or NLD%	Right-justified, high-order 21 bits after rounding to next multiple of 2048; allows a three-instruction sequence.
NLR' or NLR%	L% value with constant rounded to nearest multiple of 8192; allows a three-instruction sequence.
NLS' or NLS%	High-order 21 bits after rounding to nearest multiple of 2048; allows a three-instruction sequence.

On PA-RISC 1.0, the page size is 2048 bytes long; on PA-RISC 1.1, 2.0, and 2.0W the page size is 4096. The selectors L', LS', and LD' modulate by 2048, and the corresponding selectors R', RS', and RD' extract the offset relative to that address.

The distinction is whether the offset is always positive and between 0 and 0x7ff (L'-R'), always negative and between -0x800 and -1 (LD'-RD'), or between -0x400 and 0x3ff (LS'-RS'). This distinction is only important when using short addressing near a quadrant boundary, because only the left part is used to select a space register. Each pair is designed to work together just as L' and R' do in the previous example. See "Spaces" on page 39. The LR' and RR' prefixes are used for accessing different fields of a structure, allowing the sharing of the LR' computation.

For shared libraries, the field selectors T', LT', RT', Q', LRQ', and RRQ' are used in conjunction with the position-independent code options +z or +Z.

The field selectors P', LP', and RP' are used to form plabels (procedure labels) for use in dynamic calls. With position-independent code, the use of plabel values, rather than simple code addresses, is required. Refer to the *HP-UX Linker and Libraries Online User Guide* and *ELF 64 Object File Format*, <http://www.software.hp.com/STK/> for more information.

For example, to get a procedure label for foo, use the following code:

```
ADDIL    LTP'foo,%r27,%r1    ;get left portion of plabel pointer.
LDO      RTP'foo(%r1),%r4    ;add right portion to form a complete
                               ; plabel pointer.
```

The field selectors in the above example can also be written LP% and RP%.

## Parenthesized Subexpressions

The constant term of an expression may contain parenthesized subexpressions that alter the order of evaluation from the precedence normally associated with arithmetic operators. For example:

```
LABEL1-LABEL2+( (6765+(2048-1) ) /2048 ) *2048
```

contains a parenthesized subexpression that rounds a value up to a multiple of 2048.

Absolute symbols may be equated to constant terms containing parenthesized subexpressions as in the following sequence:

```
BASE      .EQU    0x200  
N_EL     .EQU    24  
SIZE     .EQU    (BASE+4) *N_EL
```

---

### NOTE

The use of parentheses to group subexpressions may cause ambiguities in statements where parenthesized register designators are also expected.

---

## Operands and Completers

Machine instructions usually require one or more operands.

These operands tell the processor what data to use and where to store the result. Operands can identify a register, a location in memory, or an immediate constant (that is, data that is coded into the instruction itself). The operation code determines how many and what kinds of operands are required.

Registers used in operands should be either predefined register symbols (with the % prefix) or user-defined register symbols defined with the .REG directive. They can also be absolute expressions. See “Registers and Register Mnemonics” on page 23 in this chapter.

The following example shows a few machine instructions with register operands:

```
SCRATCH .REG    %r18           ;define register SCRATCH
        ADD     %r3,%r7,%r4    ;r3 + r7 -> r4
        OR      %r7,%r3,%r8    ;inclusive or of r7,r3 -> r8
        COPY    SCRATCH,%r7    ;copy r18 to r7
        MTCTL   %r2,%sar       ;set shift amount register (cr11)
        MFSP    %sr4,%r10     ;fetch contents of sr4
```

Operands designating memory locations usually consist of an expression and a general register used as a base register. Some instructions also require a space register designation. In general, such operands are written in the form *expr(sr, gr)* or *expr(gr)*, as in the following examples:

```
local_off .EQU    -64
        LDW     4(%dp),%r2
        STW     %r0,local_off-4(%sp)
        LDW     0(%sr3,%r2),%r9
```

Notice that the space register can be omitted on instructions that allow short addressing, as in the STW instruction shown above.

If only one register is given, it is assumed to be the general register, and the space register field in the machine instruction is set to zero, which indicates short addressing.

The expression in a memory operand is either absolute or relocatable. Absolute expressions are meaningful when the base register contains the address of an array, record, or the stack pointer to which a constant offset

## Program Structure

### Operands and Completers

is required. Relocatable expressions are meaningful when the base register is %r0, or when the base register contains the left part of a 32-bit address as illustrated in the following example:

```
LDIL      L%glob,%r1                ;set up %r1 for STW
STW       %r9,R%glob(%r1)
```

Immediate operands provide data for the machine language instruction directly from the bits of the instruction word itself. A few instructions that use immediate operands are shown below:

```
ADDIL     L%var,%dp
LDIL      L%print,%r1
ADDI      4,%r3,%r5
SUBI      0x1C0,%r14,%ret0
```

Completers are special flags that modify an instruction's behavior. They are written in the opcode field, separated from the instruction mnemonic by a comma. The most common type of completer is a condition test. Many instructions can conditionally trap or nullify the following instruction, depending on the result of their normal operation. For example, notice the completers in the sequence below:

```
ADD,NSV   %r1,%r2,%r3
BL,N      handle_oflo,%r0
OR        %r3,%r4,%r5
```

The ,NSV in the ADD instruction nullifies the BL instruction if no overflow occurs in the addition operation, and execution proceeds with the OR instruction. If overflow does occur, the BL instruction is executed, but the ,N completer on the BL specifies that the OR instruction in its *delay slot* should not be executed.

Each class of machine instructions defines the set of completers that can be used.

These are described in the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and in *PA-RISC 2.0 Architecture*.

## Macro Processing

A macro is a user-defined word that is replaced by a sequence of instructions. Including a macro in a source program causes the sequence of instructions to be inserted into the program wherever the macro appears.

A user may define a word as a macro by using the `.MACRO` directive.

Detailed information about macro arguments, placement and redefinition of macros, nested macro definitions, and nested macro calls is in “Assembler Directives and Pseudo-Operations” on page 53.

## Defining New Instructions With Macros

If you are testing new CPUs or coprocessors, you may need to use opcodes that are unknown to the Assembler. A variant of a macro definition may be used to create a mnemonic for the instruction. After being defined, the new mnemonic instruction can be invoked as easily as a standard instruction.

Opcodes, subopcodes, completers, and operands are encoded into the instruction word in a bit-intensive manner because all PA-RISC instructions are one word, or 32-bits, in length.

To write a macro, you must specify explicitly which bit fields are to contain constants and which are to contain macro arguments. The macro processor has no built-in knowledge of instruction formats. Defining new instructions through macros is only possible because a convenient way to delimit bit fields has been provided. It is up to the programmer to choose the correct bit field.

Bit positions within the 32-bit word are numbered from zero to 31, from left to right. A bit range is indicated by the starting bit position followed by the ending bit position. The two bit positions are separated by two periods and enclosed in braces. The bit field beginning at bit position 6 and ending at bit position 10 is represented as:

```
{6..10}
```

If the bit field being assigned from is bigger than the bit field being assigned to, then a warning is issued and the assigned-from bit field is truncated on the left. When no bit field is specified for the assigned-from

Program Structure  
Macro Processing

value, low-order bits are used until the value of the assigned-from bit field becomes the same as the width of the assigned-to bit field. The assigned-to bit field must always be specified.

No sign extension is provided by the macro assembler when bit fields are generated.

The following macro definition defines the macro `PACK` with four formal parameters.

```
PACK      .MACRO                BASE,GREG,SREG,OFFSET
          {0..5}=0x3E{26..31}
          {6..10}=BASE{27..31}
          {11..15}=GREG{27..31}
          {16..17}=SREG{30..31}
          {18..31}=OFFSET{18..31}
          .ENDM
```

The following explanation assumes that `PACK` is invoked with the statement:

```
PACK      %sp,%r19,%sr0,-52
```

Bit Field	Description
{0..5}	Contains the six low-order bits of the new opcode <code>0x3E</code> , or binary <code>111110</code> , entered as a constant in the macro definition.
{6..10}	Contains general register 30, or binary <code>11110</code> . These are the five low-order bits of the argument <code>BASE</code> in the macro definition.
{11..15}	Contains general register 19, or binary <code>10011</code> . These are the five low-order bits of the argument <code>GREG</code> in the macro definition.
{16..17}	Contains space register 0 and represents the five low-order bits of the argument <code>SREG</code> in the macro definition.
{18..31}	Contains binary <code>1111111001100</code> , the <code>OFFSET</code> value <code>-52</code> , which was entered as an argument to the macro definition.

---

## 3

# HP-UX Architecture Conventions

The Assembler is a flexible tool for writing programs, but every operating system imposes certain conventions and restrictions on the programs that are intended to run on that system. This chapter discusses the conventions that must be understood in order to write assembly language programs and procedures for the PA-RISC instruction set on the HP 9000 Series 700 and 800 HP-UX operating system. Several Assembler directives are mentioned in this chapter to place them in a meaningful context. A full discussion of these directives is in Chapter 4, "Assembler Directives and Pseudo-Operations," on page 53.

---

## Spaces

Virtual addressing on PA-RISC is based on *spaces*. A virtual address is composed of a space identifier, which is either 16 or 32 bits long (depending on the hardware model), and a 32-bit offset within the space. Therefore, each space can contain up to 4 gigabytes, and there is a large supply of spaces.

---

### NOTE

In the 64-bit mode architecture each application is provided a flat virtual address space of  $2^{**} 64$  bytes, which is divided into four quadrants. Each quadrant is mapped into this global virtual address space by means of four space registers, which are under the control of the operating system.

Every program on an HP-UX system is assigned two spaces when it is loaded for execution by the operating system: one for code, and one for data. The HP-UX operating system makes the code space *read only*, so that it can be shared whenever several processes are executing the same program. The data space is writable by the new process, and is private to that process; that is, every process has a unique data space. The actual space identifiers assigned to these two spaces can vary from one execution of the program to the next; these numbers cannot be determined at compile time or link time. Generally, programmers do not need to be concerned with the space identifiers, since the operating system places them in two reserved space registers, where they remain

## Spaces

for the duration of program execution. The identifier of the code space is placed in space register 4 (`%sr4`) and the identifier of the data space is placed in space register 5 (`%sr5`).

When writing an assembly language program, declare a space named `$TEXT$` for executable code, and a space named `$PRIVATE$` for modifiable data. Constant data or literals that you do not plan to modify during program execution, can be placed in either space. Placing constant data in the `$TEXT$` space decreases the size of the nonsharable part of your program and improves the overall efficiency of the operating system.

The particular space registers mentioned above play an important role in virtual addressing. While many of the branching instructions, such as `BL`, `BLR`, and `BV`, are capable of branching only within the currently executing code space (called *PC-space*), two of the branching instructions, `BE` and `BLE`, require that you specify a space register as well as an offset. These instructions allow you to branch to code executing in a different space. On HP-UX systems, normally all code for a program is contained in one space, so all `BE` and `BLE` instructions should be coded to use `%sr4`.

In contrast, the memory reference instructions, such as `LDW` and `STW`, allow a choice between two forms of addressing: long and short. With long addressing, you can choose any of the space registers 1 through 3 for the space identifier part of the virtual address. The space offset is formed as the sum of an immediate displacement and the contents of a general register. With short addressing, one of the space registers between 4 through 7 is chosen automatically, based on the high-order two bits of the base register. Each space addressed by these four space registers is effectively divided into four *quadrants*, with a different quadrant of each space accessible via short addressing.

On HP-UX systems, all of a program's code is placed in quadrant zero of the `$TEXT$` space, or `%sr4`, (space offsets from 0 through `0x3FFFFFFF`). The data is placed in quadrant one of the `$PRIVATE$` space, or `%sr5` (space offsets from `0x40000000` through `0x7FFFFFFF`). Therefore, literal data in the code space and modifiable data in the data space can be addressed using the short addressing technique, without any concern for the space registers.

The identifier for shared memory segments, including shared library text, is placed into space register 6 (`%sr6`). Shared memory and shared library text are placed into quadrant two of the shared memory space (offsets `0x80000000` through `0xBFFFFFFF`). The identifier for system



code is placed into space register 7 (%sr7). System code is placed into quadrant three of the system space (offsets 0xC0000000 through 0xFFFFFFFF). Table 3-1 on page 41 shows the memory layout on HP-UX.

**Table 3-1 Memory Layout on HP-UX**

	<b>%sr4</b>	<b>%sr5</b>	<b>%sr6</b>	<b>%sr7</b>
0x00000000	Program code			
0x40000000		Program data stack Shared library data		
0x80000000			Shared memory Shared library text	
0xC0000000				System code

You can define spaces other than \$TEXT\$ and \$PRIVATE\$ in a program file by declaring a special kind of space called an *unloadable space*. Unloadable spaces are treated as normal spaces by the linker, but as the name implies, are not actually loaded when a program is executed. Unloadable spaces are typically used by compilers to store extra information within a program file. The most common example of an unloadable space is \$DEBUG\$, which is used to hold symbolic debugging information.

The *sort key* attribute allows the programmer to control the placement of a space relative to the other spaces. The linker places spaces with lower sort keys in front of spaces with higher sort keys.

The .SPACE directive is used to declare spaces. The assembly language programmer is not required to fill one space before beginning another. When a space is first declared, the Assembler begins filling that space. The .SPACE directive can also be used to return to a previously declared space, and the Assembler continues to fill it as if there had been no intervening spaces.

## Subspaces

While a space is a fundamental concept of the architecture, a subspace is just a logical subdivision of a space. The Assembler places the program's code and data into subspaces within spaces. Each subspace *belongs* to the space that was current when the subspace was first declared. The linker groups subspaces into spaces as it builds an executable program file. For more details see the *ld(1)* entry in the *HP-UX Reference*. When the linker combines several relocatable files, it groups the subspaces from each file by name, so that all subspaces with the same name are placed contiguously in the program.

## Attributes

Subspaces have several attributes. The *alignment* attribute specifies what memory alignment (in bytes) is required in the virtual address space. The alignment can be any power of two, from 1 through 4096, inclusive. Typically, the alignment is 4 or 8 to specify that the beginning of the subspace must be word or double-word aligned. Normally, the alignment attribute is computed automatically by the Assembler from the largest `.ALIGN` directive used within the subspace.

The *quadrant* attribute assigns the subspace to one of the four quadrants of its space. On HP-UX systems, all subspaces in the code space must be in quadrant 0, and all subspaces in the data space must be in quadrant 1.

The *access rights* attribute specifies the access rights that should be given to each physical page in the subspace. On HP-UX systems, all subspaces in the code space must have access rights of `0x2C` (code page executable at any privilege level). All subspaces in the data space must have access rights of `0x1F` (data page readable and writable at all privilege levels).

The *sort key* attribute allows the programmer to control the placement of a subspace relative to the other subspaces in its space. The linker places subspaces with lower sort keys in front of subspaces with higher sort keys.

## Directives

The `.SUBSPA` directive is used to declare a subspace and its attributes. As with spaces, the assembly language programmer can switch from one subspace to another, and the Assembler will fill each subspace independently as if the source code had been presented one complete subspace at a time. When the `.SPACE` directive is used to switch spaces, the Assembler remembers the current subspace in each space.

Several additional Assembler directives are provided as shorthand to declare and switch to some standard spaces and subspaces. For example, the `.CODE` directive switches to the `$TEXT$` space and the `$CODE$` subspace, and the `.DATA` directive switches to the `$PRIVATE$` space and the `$DATA$` subspace.

You can declare as many subspaces as you can use, but the sort key attribute should be used carefully, because the stack unwind mechanism reserves a range of sort keys 56 through 255 for the `$TEXT$` space. Refer to “Compiler Conventions” on page 47 in this chapter. Some of the standard subspaces and sort keys used by the compilers are shown in Table 3-2 on page 43. Directives that generate commonly used spaces and subspaces are found in Table 4-3 on page 116.

**Table 3-2 Standard Subspaces and Sort Keys**

Space	Subspace	Sort Key	Use
<code>\$TEXT\$</code>		8	
	<code>\$CODE\$</code>	24	Normal code.
	<code>\$LIT\$</code>	16	Literals.
	<code>\$MILLICODE\$</code>	8	Millicode library routines.
	<code>\$SHLIB_INFO\$</code>	0	Shared library information.
	<code>\$UNWIND\$</code>	64	Unwind information.
<code>\$PRIVATE\$</code>		16	
	<code>\$BSS\$</code>	82	Uninitialized data and common.
	<code>\$DATA\$</code>	16	Global arrays and structures.
	<code>\$DLT\$</code>	39	Data linkage table.

HP-UX Architecture Conventions  
Sections in 64-bit Mode

Space	Subspace	Sort Key	Use
	\$GLOBAL\$	40	Global variable base address.
	\$PLT	6	Procedure linkage table.
	\$SHLIB_DATA\$	12	Shared library data.
	\$SHORTBSS\$	80	Uninitialized data and common.
	\$SHORTDATA\$	24	Global scalar variables.
\$THREAD_SPECIFIC\$		16	
	\$TBSS\$	40	Thread local storage

---

## Sections in 64-bit Mode

In 64-bit mode, the Executable and Linking Format (ELF) uses segments and sections rather than spaces and subspaces.

The concept of spaces maps to the ELF concept of segments, but segments do not apply to relocatable object files. Hence, the Assembler ignores the `.SPACE` directive for 64-bit assembly programs. Subspaces map directly to the ELF concept of sections, so the `.SUBSPA` directive switches to or creates a new section. The attributes of a subspace correspond to section attributes as follows:

- Subspace names listed in the table are mapped to their corresponding section name. Names not in this table are unchanged.

SUBSPACE NAME	SECTION NAME
\$BSS\$	.bss
\$CODE\$	.text
\$DATA\$	.data
\$FINI\$	.fini

<b>SUBSPACE NAME</b>	<b>SECTION NAME</b>
\$INIT\$	.init
\$LIT\$	.rodata
\$MILLICODE\$	.text
\$PREINIT\$	.preinit
\$SHORTBSS\$	.sbss
\$SHORTDATA\$	.sdata
\$TBSS\$	.tbss

- The assembler translates access rights into a set of read, write or execute permissions for the section.
- The assembler ignores the sort key and quadrant attributes.
- The alignment attribute maps directly to the section alignment.
- The `COMMON` and `DUP_COMM` attributes map to a `COMDAT` section.
- The `CODE_ONLY`, `FIRST`, and `FROZEN` attributes are ignored.
- The `UNLOADABLE` attribute maps to a non-allocated section.

For more information about ELF, see *ELF 64 Object File Format*, at URL: <http://www.software.hp.com/STK/>.

## Location Counters

Just as spaces can be divided into subspaces, subspaces can be further divided by using location counters. You can use up to four location counters in each subspace, and the Assembler fills a separate area for each location counter. When the assembly is complete, the subspace is formed by concatenating each of these areas. All references relative to a location counter are relocated so that they are relative to the complete subspace.

Unlike subspaces, however, the use of location counters is completely local to the Assembler. Once the subspace is formed at the end of the assembly, the distinction among the individual areas built by location counters disappears. No further reordering or grouping related to location counters is performed by the linker.

This facility allows you to assemble related data into disjoint pieces of a subspace, while keeping the source code in a convenient order.

The `.LOCCT` directive is used to switch from one location counter to another. The Assembler automatically remembers the previous value of each location counter within each subspace. When the `.SUBSPA` directive is used to switch subspaces, the Assembler automatically begins using the location counter that was last in effect in the new subspace.

## Compiler Conventions

In order to write assembly language procedures that can both call to and be called from high-level language procedures, it is necessary to understand the standard procedure-calling convention and other compiler conventions.

On many computer systems, each high-level language has its own calling convention. Consequently, calls from one language to another are sometimes difficult to arrange, except through assembly code. The architecture generally prescribes very few operations that must be done to effect a procedure call, and there is often a pair of machine-language instructions to call a procedure and return from one. PA-RISC architecture provides no special procedure call or return instructions.

There is, however, a standard procedure-calling convention for all high-level languages as well as the Assembler. It is tuned for the architecture, and is designed to make a procedure call with as few instructions as possible.

Besides defining a uniform call and return sequence for all languages, the calling convention is important for other reasons. In order to streamline the calling sequence, the return link is not saved on the stack unless necessary and the previous stack pointer is rarely saved on the stack. Therefore, it is not usually possible to obtain a stack trace at an arbitrary point in the program without some additional static information about each procedure's stack frame size and usage.

For example, you could not obtain a stack trace while debugging or analyzing a core dump, or using the `TRY/RECOVER` feature in HP Pascal/HP-UX. Obtaining a stack trace is made possible by the *stack unwind* mechanism. It uses special *unwind descriptors* that contain the exact static information needed for each procedure. These descriptors are generated automatically by the linker based on information provided by all high-level compilers as well as the Assembler.

Each descriptor contains the starting and ending address of a procedure's object code, plus that procedure's stack frame size, and a few flags indicating, among other things, whether the return link is saved on the stack. Given the current program counter and stack pointer, the stack unwind mechanism can determine the calling procedure by finding

## Compiler Conventions

the return link either in a register or on the stack. Also, it can determine the previous stack pointer by subtracting the current procedure's stack frame size.

The Assembler requires that you follow programming conventions to generate unwind descriptors. The beginning and end of each procedure must be noted with the `.PROC` and `.PROCEND` directives. The `.CALLINFO` directive supplies additional information about the procedure, including the stack frame size. The Assembler passes this information to the linker, which creates the unwind descriptor. It can also generate the standard entry and exit code to create and destroy the stack frame, save and restore the return link (if necessary), and save and restore any necessary registers. These code sequences are generated at the points indicated by the `.ENTER` and `.LEAVE` pseudo-operations. For a more thorough discussion of programming conventions, refer to the *64-bit Runtime Architecture for PA-RISC 2.0*, at URL: <http://www.software.hp.com/STK/>.

Arguments to procedures are loaded into general registers 26, 25, 24, and 23; these registers are named, respectively, `%arg0`, `%arg1`, `%arg2`, and `%arg3`. If more than four words of arguments are required, the remaining arguments are stored in the caller's stack frame in the variable argument list. The return value should be returned in general register 28, called `%ret0`. General register 29, called `%ret1`, is used for the low-order bits of a double-word return value, while `%ret0` contains the high order bits. In addition to the argument and return registers, the procedure can use registers 19 through 22 and registers 1 and 31 as scratch registers. Any other general registers must be saved before use at entry and restored before exit.

Chapter 4, "Assembler Directives and Pseudo-Operations," on page 53 contains detailed descriptions of the Assembler directives described above. For a more thorough discussion of the procedure calling conventions, refer to the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>.

In order for an assembly language procedure to be callable from another language or another assembly language module, the name of the procedure must be *exported*. The `.EXPORT` directive does this. It also allows you to declare the *symbol type*. For procedure entry points, the symbol type should be `ENTRY`.

The Assembler and linker treat all symbols as case-sensitive, while some compilers do not. By convention, compilers that are case-insensitive uniformly convert all exported names to lower case. For example, it is



possible to declare a procedure that cannot conflict with HP Pascal/HP-UX procedure names by using uppercase letters. However, there is an *aliasing* mechanism in some compilers that allows you to declare a case-sensitive name for external use. See the appropriate language reference manual for more information.

Conversely, the `.IMPORT` directive allows you to reference a procedure name that is exported from another module, either from the Assembler or the compiler. Once a procedure name has been imported, it can be referenced exactly as if it were declared in the same module.

Data symbols can be exported and imported just like procedure names. However, not all compilers export the names of global variables, or provide a mechanism to reference data symbols exported from an assembly language module. For example, the HP Pascal/HP-UX compiler does not normally do this, while the HP C/HP-UX compiler does. HP FORTRAN 77/HP-UX named common blocks are exported, but the names of the variables within the common blocks are not.

It was mentioned before that data is allocated beginning from a virtual space offset `0x40000000`. For convenience as well as compatibility with future releases of HP-UX systems, all data in the `$PRIVATE$` space must be accessed relative to general register 27, called `%dp`. EStandard run-time start-up code, from the file `/usr/ccs/lib/crt0.o`, must be linked with every program. This start-up code declares a global symbol called `$global$` in the `$GLOBAL$` subspace. This code also loads the address of this symbol into the `%dp` register before beginning program execution. This register must not be changed during the execution of a program. Since the `%dp` register is known to contain the address of `$global$`, the following single instruction does the load as long as the displacement from `$global$` to the desired location is less than 8 kilobytes:

```
LDW      var-$global$(%dp),%r3
```

If the desired location is not known to be close enough to `$global$`, use the following sequence:

#### Global Symbol Usage

```
ADDIL    L'var-$global$,%dp          ;result in r1
LDW      R'var-$global$(%r1),%r3
```

**Compiler Conventions**

For convenience, the `$$SHORTDATA$` and `$$SHORTBSS$` subspaces can be used for small scalar variables. Most scalar variables are close enough to `$$GLOBAL$` so that the shorter form can be used. Arrays and large structures should be defined in `$$DATA$` and the long form used.

To access items in the `$$PRIVATE$` space (global data), the following does not work:

```
LDIL      L'var,%r1                ;wrong
LDW       R'var(%r1),%r3          ;wrong
```

This example assumes that the operating system always allocates data at the same virtual space offset `0x40000000`.

Thread local storage (TLS) data is accessed relative to control register 27 (`%cr27`). The contents of `%cr27` must first be moved to a general register by using the `MFCTL` instruction. A symbol, `__tp`, is defined, similar to `$$global$`. The following code shows the loading of the TLS variable. Note the similarities between this example and the example “Global Symbol Usage” on page 49.

```
MFCTL     %cr27, &rX
ADDIL     L'var-__tp,%rX          ;result in r1
LDW       R'var-__tp(%r1),%r3
```

Uninitialized areas in the data space can be requested with the `.COMM` (common) request. These requests are always made in the `$$BSS$` subspace in the `$$PRIVATE$` space. The `$$BSS$` subspace should not be used for any initialized data. Common requests are passed on to the linker, which matches up all requests with the same name and allocates a block of storage equal in size to the largest request. If, however, an exported data symbol is found with the same name, the linker treats the common requests as if they were imports.

HP FORTRAN 77/HP-UX common blocks are naturally allocated in this way: if a `BLOCK DATA` subprogram initializes the common block, all common requests are linked to that initialized block. Otherwise, the linker allocates enough storage in `$$BSS$` for the common block. The HP C/HP-UX compiler also allocates uninitialized global variables this way. In C, however, each uninitialized global is a separate common request.

---

## Shared Libraries

The field selectors `T'`, `LT'`, and `RT'` are used to write position-independent code in assembly language. When you use these selectors and invoke the Assembler with the `as` command, you must use the `+z` or `+Z` compiler option on the command line.

Any assembly code that is to be used with shared libraries must follow the standard procedure call mechanism as defined in the runtime architecture documents under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>. Any external procedures must be exported as type `ENTRY` for the shared library interface to work correctly.

For more information on position-independent code and shared libraries, refer to the *HP-UX Linker and Libraries Online User Guide* and the *ELF 64 Object File Format*, URL: <http://www.software.hp.com/STK/>.

---

## Assembly Listing

The Assembler command-line option, `-l`, causes an assembly listing to standard output. For each line of source code, the listing provides:

- line number
- the subspace offset
- the hexadecimal representation of the assembled code (possibly flagged with an asterisk (\*) to indicate address relocation)
- the source text
- any comments.

The following is a line of assembly language as it appears in the source file:

```
SAVE      LDO      VAL(%r0),%r20      ;retain value
```

The above line would appear in the assembly listing as follows:

line no.	offset	hex representation	label	opcode	operands	comment
16	0000004c	(341400A)	SAVE	LDO	VAL(%r0),%r20	;retain value

## Assembly Listing

The choice of line number 16 is arbitrary here. At the end of the assembly listing, a symbol table is printed showing the name and value of each symbol in the file. A type field for each symbol, indicating either absolute or relocatable, is included.

Certain types of source lines generate multiple instructions. Macro calls often expand to several instructions. The `.ENTER` and `.LEAVE` pseudo-operations can each generate more than one instruction. The predefined subspace directives, such as `.CODE` and `.DATA`, result in a space and a subspace declaration.

You have the choice of listing a section of assembled code in either the compressed or expanded form. The placement of the `.LISTON` and `.LISTOFF` directives determines which code will be expanded during listing. The directive `.LISTON` tells the Assembler to expand the listing of all subsequent source lines until a `.LISTOFF` directive is encountered. `.LISTOFF` stays in effect until the occurrence of a `.LISTON` directive.

The default is `.LISTON`.

The directives `.LISTON` and `.LISTOFF` may be placed anywhere in the source text and always go into effect immediately. The `.LISTON` and `.LISTOFF` directives can be used as often as desired.

---

## 4 Assembler Directives and Pseudo-Operations

Assembler directives and pseudo-operations allow you to take special programming actions during the assembly process. The directive and pseudo-operation names begin with a period (.) to distinguish them from machine instruction opcodes or extended opcodes.

---

### Introduction

Table 4-1 lists the Assembler directives. Table 4-2 on page 55 lists the pseudo-operations. The directives include those that establish the procedure-calling convention, declare common, and define spaces and subspaces. The pseudo-operations reserve and initialize data areas.

The remainder of this chapter lists the Assembler directives and pseudo-operations in alphabetic order. Several of the descriptions include sample assembly code sequences. You can enter these short code sequences, assemble them using the `-l` option of the `as` command, then inspect the offsets and field values to see how that particular directive controls the assembly environment.

This chapter also includes Table 4-3 on page 116 under “Programming Aids” on page 116, which lists the predefined directives that establish standard spaces and subspaces.

**Table 4-1**      **Assembler Directives**

<b>Directive</b>	<b>Function</b>
<code>.ALIGN</code>	Forces location counter to the next largest multiple of the supplied alignment value.
<code>.ALLOW</code>	Used with a <code>.LEVEL</code> directive, it temporarily allows the use of features in the architecture specified in the <code>.LEVEL</code> directive.
<code>.CALL</code>	Specifies that the next statement is a procedure call.

## Assembler Directives and Pseudo-Operations

<b>Directive</b>	<b>Function</b>
.CALLINFO	Provides information for generating Entry/Exit code sequences and for creating stack unwind descriptors.
.COMM	Requests common storage for a specified number of bytes.
.COPYRIGHT	Inserts a string into the object module as a copyright notice.
.END	Terminates an assembly language program.
.ENDM	Marks the end of a macro definition.
.ENTRY	Marks the entry point of the current procedure.
.EQU	Assigns an expression to an identifier.
.EXIT	Marks the return point of the current procedure.
.EXPORT	Makes a specified symbol available to other modules.
.IMPORT	Specifies that the definition of the given symbol occurs in another module.
.LABEL	Permits a label definition to appear within a sequence of directives that occur on a single line.
.LEVEL	Makes the object file a PA-RISC 1.1, 2.0, or 2.0W file.
.LISTOFF	Controls listing of expanded Assembler instructions.
.LISTON	Controls listing of expanded Assembler instructions.
.LOCCT	Selects a location counter.
.MACRO	Marks the beginning of macro definitions.
.ORIGIN	Advances the location counter to a relative location from the beginning of the current subspace.

<b>Directive</b>	<b>Function</b>
.PROC	Marks the first statement in a procedure.
.PROCEND	Marks the last statement in a procedure.
.REG	Attaches a type and number to a user-defined register name.
.SHLIB_VERSION	Inserts a date string into the object module as a shared-library version identifier.
.SPACE	Declares a new space or switches back to a previous space.
.SUBSPA	Declares a new subspace or switches back to a previous subspace.
.VERSION	Inserts the specified string into the current object module as a user-defined version identification string.

**Table 4-2****Pseudo-Operations**

<b>Directive</b>	<b>Function</b>
.BLOCK	Reserves a block of data storage.
.BLOCKZ	Reserves a block of data storage.
.BYTE	Reserves 8 bits (a byte) of storage and initializes it to the given value.
.DOUBLE	Initializes 64 bits (a double-word) of storage to a floating-point value.
.DWORD	Reserves 64 bits (a double word) of storage and initializes it to the given value.
.ENTER	Marks a procedure's entry point and generates standard entry code.
.FLOAT	Initializes a single-word of storage to a floating-point value.

## Assembler Directives and Pseudo-Operations

<b>Directive</b>	<b>Function</b>
. HALF	Reserves 16 bits (a half word) of storage and initializes it to the given value.
. LEAVE	Marks a procedure's exit point and generates standard exit code.
. SPNUM	Reserves and initializes a word of storage.
. STRING	Reserves the appropriate amount of storage and initializes it to the given string.
. STRINGZ	Reserves the appropriate amount of storage and initializes it to the given string.
. WORD	Reserves 32 bits (a word) of storage and initializes it to the given value.



---

## .ALIGN Directive

The `.ALIGN` directive advances the current location counter to the next specified “boundary.”

### Syntax

`.ALIGN [ boundary]`

### Parameters

**boundary**      An integer value for the byte boundary to which you want to advance the location counter. The Assembler advances the location counter to that boundary. Permissible values must be a power of 2 and can range from one to 4096. The default value is 8 (double word aligned).

### Example

This sample program adds a 21 bit field to the data pointer. Then a branch is taken to the label “page” that has been page-aligned.

```
.CODE
ADDIL L'$WORDMARK$-$global$',%dp
B page
NOP
.ALIGN 4096
page
ADDI 1,%r1,%r1
.DATA
$WORDMARK$
.WORD 0x0FFF
.IMPORT $global$,DATA
```

## **.ALLOW Directive**

The `.ALLOW` directive tells the Assembler to temporarily allow PA-RISC features from a higher version level of the PA-RISC architecture. The `.ALLOW` directive also tells the Assembler to temporarily allow implementation-specific features in the assembly source file.

### **Syntax**

`.ALLOW 1.1`

Lines of source code

`.ALLOW`

### **Parameters**

- 1.1            Allows PA-RISC 1.1 features.
- 2.0            Allows PA-RISC 2.0 features.

### **Discussion**

Use the `.ALLOW` directive with the `.LEVEL` directive. The Assembler uses the `.LEVEL` directive to mark the relocatable object file with the proper PA-RISC architecture version level. In the source file, the Assembler emits warning messages whenever a feature is used that is not appropriate for the specified `.LEVEL` directive.

Use the `.ALLOW` directive when it is necessary to include features or instructions from a later version of PA-RISC while leaving the relocatable object file marked as an earlier PA-RISC architecture version. For example, use the `.ALLOW` directive when you need to include PA-RISC 2.0 features or instructions while leaving the relocatable object file marked as a PA-RISC 1.1 architecture version.

---

**NOTE**

---

A `2.0W` parameter is *not* permitted with `.ALLOW`, because the code generated for `2.0W`(64-bit mode) is incompatible with other levels.

When using the `.ALLOW` directive, a run-time check must be inserted into the assembly source code. This run-time check should insure that the code is executing on a PA-RISC processor that supports the feature or features being used after the `.ALLOW` directive. See the example below.

An `.ALLOW` directive without a parameter signals the end of the region that the previous `.ALLOW` directive was controlling. Control is returned to the `.LEVEL` specified for the file.

---

**NOTE**

---

The `.ALLOW` and `.LEVEL` directives replace the `+DA` and `+DS` command-line compiler options.

## Example

The following example shows how to set a range of memory to 0. In PA-RISC 1.1 architecture, use the `stw` instruction. In PA-RISC 2.0 architecture, use the more efficient `std` instruction.

```
.LEVEL 1.1
; This object file will be marked as a PA 1.1 object file

; Check what version of PA Architecture we are linked for
addil LR'_SYSTEM_ID-$global$,%dp
ldw RR'_SYSTEM_ID-$global$(%r1),%r5
ldi CPU_PA_RISC1_1,%r4
combt,<,n %r4,%r5,$00000002
; 1.1 specific code
$00000001
addib,< 1,%r23,$00000001
stw,ma %r0,4(%r31)
b,n $00000003
; 2.0 specific code
$00000002
.ALLOW 2.0
addib,< 2,%r23,$00000002
std,ma %r0,8(%r31)
.ALLOW
$00000003
; General code
```

## **.BLOCK and .BLOCKZ Pseudo-Operations**

The `.BLOCK` and `.BLOCKZ` pseudo-operations reserve a block of storage.

### **Syntax**

`.BLOCK [ num_bytes]`

`.BLOCKZ [ num_bytes]`

### **Parameters**

*num\_bytes*      An integer value for the number of bytes you want to reserve. Permissible values range from zero to `0x3FFFFFFF`. The default value is zero.

### **Discussion**

The `.BLOCK` pseudo-operation reserves a data storage area but does not perform any initialization. The `.BLOCKZ` pseudo-operation reserves a block of storage and initializes it to zero.

When you label a `.BLOCK` pseudo-operation, the label refers to the first byte of the storage area.

For large blocks, it is usually better to use the `.COMM` directive to allocate uninitialized space. Since `.COMM` storage is allocated at run time, it doesn't increase the size of the object file.

---

#### **NOTE**

Under the present implementation of the Assembler, the `.BLOCK` pseudo-operation also initializes the reserved area to zero.

---

## Example

The first example requests the Assembler to reserve 64 bytes of memory in the `$CODE$` subspace. This area is then followed by a “Load Word” and “Store Word” instruction.

```
                .SPACE $TEXT$  
                .SUBSPA $CODE$  
                .BLOCK 64  
swap           LDW 0(%r2)%r1  
                STW %r1,4(%r2)  
                .END
```

The second example reserves 32 bytes of memory in the `$DATA$` subspace followed by one word intended as an end marker.

```
word0          .DATA  
                .BLOCK      0X20  
word8          .WORD      0XFFFF
```

---

## **.BYTE Pseudo-Operation**

The `.BYTE` pseudo-operation reserves storage and initializes it to the given value.

### **Syntax**

```
.BYTE [ init_value [ , init_value ] ...]
```

### **Parameters**

*init\_value*      Either a decimal, octal, or hexadecimal number or a sequence of ASCII characters, surrounded by quotation marks. If you omit the initializing value, the Assembler initializes the area to zero.

### **Discussion**

The `.BYTE` pseudo-operation requests 8 bits of storage. If the location counter is not properly aligned on a boundary for a data item of that size, the Assembler advances the location counter to the next multiple of that item's size before reserving the area.

When you label the pseudo-operation, the label refers to the first byte of the storage area. Operands separated by commas initialize successive units of storage.

### **Example**

The first pseudo-operation allocates a byte labeled `E` and initializes it to the character `[ .`

```
E        .BYTE        " [ "
```

---

## .CALL Directive

The `.CALL` directive marks the next branch statement as a procedure call, and permits you to describe the location of arguments and the function return result.

### Syntax

```
.CALL [ argument_description[ argument_description] ...]
```

### Parameters

*argument\_*  
*description*

Allows you to communicate to the linker the types of registers used to pass floating point arguments and receive floating point return results in the succeeding procedure call. Similarly, this information can be communicated in the `.EXPORT` directive.

The linker requires this information because the runtime architecture allows floating point arguments and return values to reside in either general registers or floating point registers, depending on source language convention. At link time, the linker ensures that both the caller and called procedure agree on argument location. If not, the linker may insert code to relocate the arguments (or return result) before control is transferred to the called procedure or a procedure return is completed.

You can use up to 5 *argument-descriptions* in the `.CALL` directive; one for each of the four arguments that may be passed in registers (`arg0-arg3`), and one for a return value (`ret0`).

---

#### NOTE

In PA-RISC 2.0W, (64-bit mode) the Assembler ignores the `.CALL` directive. This means that the linker does not ensure that the caller and called procedure agree on argument locations. If you do not know the prototype of the called procedure, you must pass floating point

**.CALL Directive**

parameters in both the corresponding general registers and corresponding floating-point registers. See the documents under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>.

---

The form of *argument-description* is:

*arg=location*

where *arg* can be:

ARGW0	The first word in the argument list.
ARGW1	The second word in the argument list.
ARGW2	The third word in the argument list.
ARGW3	The fourth word in the argument list.
RTNVAL	The return value for a procedure.

and *location* can be:

NO	The argument word cannot be relocated. This should be used for all nonfloating-point arguments; it is the default when an <i>argument-description</i> is omitted.
GR	The argument word occurs in a general register.
FR	The argument word occurs in a floating point register.
FU	The argument word occurs in the upper half of a floating-point register.

**Example**

This example shows the use of the `.CALL` directive in 32-bit mode.



```

; This program calls printf() with four arguments
; whose register locations are described in the .CALL directive.
; The format string goes into arg0, not to be relocated.
; The string "message" goes into arg1, specified as a general register.
; The floating-point value 57005.57005 goes into farg2,
; specified as a floating-point register.
; The hexadecimal number 0xf00d goes into arg3,
; specified as a general register.
; The return value from printf() is not to be relocated.

        .LIT
        .ALIGN 8
        .WORD 1197387154          ; floating-point literal
fp2     .BLOCKZ 12
        .WORD 0
        .CODE
main    .PROC
        .CALLINFO CALLER,FRAME=24,SAVE_RP
        .ENTER
LDIL   L'fp2,%r1
LDO    R'fp2(1),%r31      ; r31 < - floating-point literal address
FLDWS  -16(%r31),%fr4
LDO    -64(%sp),%r19
FSTWS  %fr4,0(%r19)
ADDIL  L'61453,0
LDO    R'61453(%r1),%r20
STW    %r20,-68(%sp)    ; end of stacking floating-point address
ADDIL  L'string_area-$global$,%dp
LDO    R'string_area-$global$(%r1),%r21    ; point to "message"
STW    %r21,-60(%sp)    ; stack "message" address
LDO    -64(%sp),%r22
FLDWS  0(%r22),%fr5
FCNVFF,SGL,DBL    %fr5,%fr6    ; convert floating-point value
ADDIL  L'string_area-$global$+8,%dp
LDO    R'string_area-$global$+8(%r1),%arg0
                                ;point to format string
LDW    -60(%sp),%arg1    ; load "message" argument
FSTDS  38,-16(%sp)
FLDWS  -12(%sp),%fr6    ; load floating-point argument
LDWS   -16(%sp),%arg3    ; load hexadecimal argument
LDW    -68(%sp),%r1
STW    %r1,-52(%sp)
        .CALL  argw0=no,argw1=gr,argw2=fr,argw3=gr,rtnval=no
        BL    printf,2
        NOP
        .LEAVE

```

## Assembler Directives and Pseudo-Operations

### .CALL Directive

```
.PROCEND
.EXPORT main,ENTRY
.IMPORT printf,CODE

.DATA
string_area
.ALIGN 8
.STRINGZ      "message"
.STRINGZ      "ARGS = %s,%f,%x\n"
.IMPORT $global$,DATA
```

---

## .CALLINFO Directive

.CALLINFO is a required directive that describes the environment of the current procedure. The information it provides is available to the .ENTER and .LEAVE pseudo-operations to control the entry and exit code sequences that they generate. Additional information is used by the Assembler to direct the creation of stack unwind descriptors.

### Syntax

.CALLINFO [ *parameter* [ , *parameter* ] ...]

where *parameter* is one of:

ALLOCA\_FRAME  
ARGS\_SAVED  
CALLER  
CALLS  
NO\_CALLS  
CLEANUP  
ENTRY\_FR=*number*  
ENTRY\_GR=*number*  
ENTRY\_SR=*number*  
FRAME=*number*  
HPUX\_INT  
MILLICODE  
NO\_UNWIND  
SAVE\_MRP  
SAVE\_RP  
SAVE\_SP  
SAVE\_SR0

### Parameters

**ALLOCA\_FRAME** Indicates that this procedure allocates temporary storage by modifying the stack pointer (%r30). A copy of the frame pointer is normally placed in %r3. However, if this procedure also has a large frame (FRAME > 8191), then the copy of the frame pointer is placed in %r4 instead.

**ARGS\_SAVED** Indicates that this procedure stores the arguments into the stack frame.

**.CALLINFO Directive**

**CALLER or  
CALLS**

Indicates that this procedure calls other routines, so it requires space in the stack for a frame marker and a fixed argument list. (When a program is assembled using the `-f` option, this becomes the default case.)

The Assembler allocates stack space when it encounters an `.ENTER` pseudo-operation and deallocates this space when it encounters a `.LEAVE` pseudo-operation. The Assembler allocates 48 bytes for the PA-RISC 32-bit mode and 80 bytes for the PA-RISC 64-bit (2.0W) mode.

The frame marker and fixed argument list area occur at the top of the stack so you must take this space into account when locating local variables on the stack. You must allocate an area (using `FRAME=`) for a variable argument list when this area is needed.

`CALLER` does not imply the existence of the parameter `SAVE_RP`.

The `CALLER` and `CALLS` parameters are equivalent.

**NO\_CALLS**

Indicates that the procedure does not call other procedures and, therefore, does not require a frame marker on the stack. This is the default case unless the program is assembled using the `-f` option.

**CLEANUP**

Indicates that this procedure requires cleanup during unwind.

**ENTRY\_FR=  
*register***

Specifies the high end boundary of the Entry/Save floating-point register partition. The partition includes `%fr12` through `%fr15` for PA-RISC 1.0 and `%fr12` through `%fr21` for PA-RISC 1.1. The Assembler automatically saves these registers when it encounters an `.ENTER` pseudo-operation and restores them when it encounters a `.LEAVE` pseudo-operation.

**ENTRY\_GR=  
*register***

Specifies the high end boundary of the Entry/Save register partition. The partition may extend over registers `%r3` through `%r18`. If you omit this parameter, no registers are saved.

When a procedure uses these registers, the Assembler saves their values when it encounters an `.ENTER` pseudo-operation and restores these values when it encounters a `.LEAVE` pseudo-operation. The called routine saves these registers upon entry and restores them upon exit, so values in Entry/Save registers are preserved across a procedure call.

**Note:** See the description of the `FRAME` parameter regarding the use of `%r3`.

`ENTRY_SR=`  
*register*

Specifies the high end boundary of the space register partition. The partition currently contains only `%sr3`. When the `.CALLINFO` directive includes this parameter, the Assembler automatically saves the Space Register when it encounters an `.ENTER` pseudo-operation and restores this register when it encounters a `.LEAVE` pseudo-operation.

`FRAME=number`

Defines the combined size (in bytes) of the local variable area and variable argument area needed by the procedure. The `.ENTER` pseudo-operation allocates the desired space for local variables below the frame marker and the `.LEAVE` pseudo-operation deallocates that space.

The *number* parameter must be a multiple of eight bytes. If a `.CALLINFO` directive lacks this parameter, the Assembler assumes a default frame size of zero.

The stack frame includes space for local variables and the variable argument area. The size specified for the frame should not include space for the stack frame marker or the fixed argument area. Allocation of these areas is controlled by the `CALLER` and `NO_CALLS` parameters. The inclusion of the `CALLER` parameter always allocates space for the stack frame marker and the fixed argument area. (See Table 4-1 on page 53)

A frame marker is required if the assembly routine calls another routine.

**.CALLINFO Directive**

For PA-RISC 32-bit mode, the frame area is offset from the Stack Pointer by 48 bytes because the frame marker contains 32 bytes and the fixed argument list contains 16 bytes, when both of these areas are present.

For PA-RISC 2.0W (64-bit mode), the frame area is offset from the Stack Pointer by 80 bytes because the frame marker contains 16 bytes and the fixed argument list contains 64 bytes.

However, the Assembler does not allocate space for the frame marker and fixed argument list if the procedure does not call any other routines (see the `NO_CALLS` parameter).

If the total frame size for a procedure exceeds 8191 bytes, the Assembler uses `%r3` to locate the previous frame marker when it encounters an `.ENTER` or `.LEAVE` pseudo-operation. Under these circumstances, changing the value of `%r3` can cause serious consequences.

<code>HPUX_INT</code>	Specifies that this procedure is an interrupt procedure. This is necessary for the stack unwind mechanism.
<code>MILLICODE</code>	Indicates to the unwind mechanism that this is a millicode routine and it should follow the millicode calling conventions.
<code>NO_UNWIND</code>	This is to be used only in the context of <i>stand-alone code</i> or any procedure that does not need to be reliably <i>unwound</i> .
<code>RP_IN_R31</code>	Indicates that the return pointer has been moved to register <code>%r31</code> in order to make local millicode calls. This parameter is only valid for the PA-RISC 2.0W (64-bit mode).
<code>SAVE_MRP</code>	Indicates that this millicode procedure saves the Millicode Return Pointer (MRP) in its frame marker at <code>(SP-20)</code> .
<code>SAVE_RP</code>	Specifies that the frame marker of the previous routine stores the value of the Return Pointer (RP). The Assembler automatically saves the Return Pointer when it encounters an <code>.ENTER</code> pseudo-operation, and

it restores the `RP` value when it encounters a `.LEAVE` pseudo-operation. Generally, any procedure that calls other routines should save the `RP` value.

`SAVE_SP` Specifies that the current routine saves the value of `Previous_SP` in its frame marker at `SP-4`. Because the Assembler does not automatically save the Stack Pointer when it generates Entry/Exit code sequences, you must explicitly save this value in your program when using this key word. You can obtain the `Previous_SP` value from the special register `%previous_sp`.

Programming languages, such as HP Pascal/HP-UX, typically use this value for up-level display pointers to reference local variables.

`SAVE_SR0` Indicates that this millicode procedure saves `%sr0` in its frame marker at `(SP-16)`. This parameter is not valid for PA-RISC 2.0W (64-bit mode).

## Discussion

When a program uses the `.CALLINFO` directive, all entry and exit code must follow the procedure calling convention described in the documents under the topic PA-RISC Architecture at URL:

<http://www.software.hp.com/STK/>. If you use the `.ENTER` and `.LEAVE` directives, the Assembler will automatically generate the necessary code. The parameters in the `.CALLINFO` directive govern the generation of the Entry/Exit code sequence (except for `SAVE_SP`). However, if you use the `.ENTRY` and `.EXIT` directives, your code must provide the necessary Entry/Exit code sequences.

A stack frame consists of a pointer to the top of the frame, a frame marker, a fixed argument list, and a variable argument list. The following example, Stack Frames, illustrates these areas as an inverted stack for PA-RISC 1.x and 2.0.

---

**NOTE**

For PA-RISC 2.0W, 64-bit mode, the stack frame is different. Refer to the documents under the topic PA-RISC Architecture at URL:  
<http://www.software.hp.com/STK/>.

---

## Assembler Directives and Pseudo-Operations

### .CALLINFO Directive

#### Stack Frames

```

                                     Variable Arguments
. . .
SP-64:      arg word 7
SP-60:      arg word 6
SP-56:      arg word 5
SP-52:      arg word 4

                                     Fixed Arguments

SP-48:      arg word 3 / ARG3
SP-44:      arg word 2 / ARG2
SP-40:      arg word 1 / ARG1
SP-36:      arg word 0 / ARG0

                                     Frame Marker

SP-32:      Saved %r19 for shared library calls.
SP-28:      Reserved
SP-24:      Saved RP for shared library calls.

SP-20:      Saved RP (or SAVED_MRP).

SP-16:      Static Link (or SAVED %sr0).
SP-12:      Clean Up.
SP-8:       Extension Pointer. Calling stub RP (RP").
SP-4:       Previous SP.

                                     Top of Frame

SP:         Stack Pointer.
```

### Example

This example uses the C `printf()` routine (see *printf(3S)* in *HP-UX Reference*). It illustrates most of the directives to be used when assembly language programmers follow the standard procedure calling conventions described in the documents under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>.

```
.CODE                                ; declare space and subspace
main
.PROC                                ; delimit procedure entry
.CALLINFO CALLER,FRAME=0,SAVE_RP    ; no local variables, need return
.ENTER                               ; insert entry code sequence
ADDIL L'stringinit-$global$,%r27    ; point to data to be printed
LDO   R'stringinit-$global$(%r1),%r26 ; place argument to printf
.CALL                               ; set up for procedure call
BL    printf,%r2                    ; call printf, remembering from where
NOP
.LEAVE                               ; insert exit code sequence
.PROCEND                             ; delimit procedure end

.DATA                                ; declare space and subspace
stringinit
.IMPORT $global$,DATA               ; get data reference point
```



## Assembler Directives and Pseudo-Operations

### **.CALLINFO Directive**

```
.CODE ; re-enter code subspace
.EXPORT main,ENTRY ; make routine known to linker
.IMPORT printf,CODE ; external procedure declaration
.END
```

---

## .COMM Directive

The `.COMM` directive makes a storage request for a specified number of bytes.

### Syntax

```
label .COMM [ num_bytes]
```

### Parameters

<i>label</i>	Labels the location of the reserved storage.
<i>num_bytes</i>	An integer value for the number of bytes you want to reserve. The Assembler uses a default value of 4 if the <code>.COMM</code> directive lacks a <i>num_bytes</i> parameter. Permissible values range from one to <code>0x3FFFFFFF</code> .

### Discussion

The `.COMM` directive declares a block of storage that can be thought of as a *common block*. You must label every `.COMM` directive. The linker associates the *label* with the subspace in which the `.COMM` directive is declared and allocates the necessary storage within that subspace.

`.COMM` always allocates its space in the `$BSS$` subspace of the `$PRIVATE$` space. If the *label* of a `.COMM` directive appears in several object modules, the linker uses the maximum size specified in any module when it allocates the necessary storage in the current subspace.

### Example

This example reserves 16 bytes of storage for `mydata`.

```
mydata    .BSS  
          .COMM 16
```

## **.COPYRIGHT Directive**

The `.COPYRIGHT` directive inserts a company name and date into the object module as a copyright notice.

### **Syntax**

```
.COPYRIGHT "company-name [ , date] "
```

### **Parameters**

*company-name*,  
*date*            A sequence of ASCII characters, surrounded by quotation marks. The string can contain up to 256 characters. When a comma follows the company name, the next text is expected to be the date.

### **Discussion**

The following is the standard copyright message placed in the copyright header of the object file:

```
Copyright company-name, date. All rights reserved. No
part of this program may be photocopied, reproduced, or
transmitted without prior written consent of
company-name.
```

---

**NOTE**

This directive can appear anywhere in the source file, but may appear only once.

---

## Example

This program places a copyright notice in the object file. Once the copyright notice is in the object file, the HP-UX utility strings can be used to access it. See *strings(1)* in *HP-UX Reference*.

```
.COPYRIGHT "My Company Name, 8 Nov 1994"  
.CODE  
.EXPORT main,ENTRY  
main  
.PROC  
.CALLINFO  
.ENTER  
LDI    2,%r5  
ADDI   2,%r5,%r6  
.LEAVE  
.PROCEND
```

## .DOUBLE Pseudo-Operation

The `.DOUBLE` pseudo-operation initializes a double-word to a floating-point value, calculated from the parameters provided. If the location counter, is not aligned on a double-word boundary, it is forced to the next multiple of eight. If the statement is labeled, the label refers to the first byte of the storage area.

### Syntax

```
.DOUBLE integer [ .fraction] [ E [ -] power]  
.DOUBLE .fraction [ E [ -] power]
```

### Parameters

<i>integer</i>	Specifies the whole number part of a decimal number.
<i>fraction</i>	Specifies the fractional part of a decimal number.
<i>power</i>	Specifies the power of ten to raise a decimal number. To raise the decimal number to a negative power of ten, place a minus sign (-) directly in front of the power specified.

### Example

Each of the following examples initializes two words of memory to floating-point quantities: 0.00106 and 400000.0 respectively.

```
dec_val1 .DOUBLE 10.6E-4  
dec_val2 .DOUBLE 0.4E6
```

---

## **.DWORD Pseudo-Operation**

The `.DWORD` pseudo-operation reserves storage and initializes it to the given value.

### **Syntax**

```
.DWORD [ init_value [ , init_value ] ...]
```

### **Parameters**

*init\_value*      An absolute expression, a decimal, octal, or hexadecimal number, or a sequence of ASCII characters surrounded by quotation marks. If you omit the initializing value, the Assembler initializes the area to zero.

### **Discussion**

The `.DWORD` pseudo-operation requests 64 bits of storage. If the location counter is not properly aligned on a boundary for a data item of that size, the Assembler advances the location counter to the next multiple of that item's size before reserving the area.

When you label the pseudo-operation, the label refers to the first byte of the storage area. Operands separated by commas initialize successive units of storage.

### **Example**

The first pseudo-operation advances the current subspace's location counter to a double word boundary, allocates a double word of storage labeled `F` and initializes that double word to minus 64 (2's complement). The second pseudo-operation initializes a double word of storage to the hexadecimal number `6efffff12345678`.

```
F     .DWORD     64
       .DWORD     0X6efffff12345678
```

## **.END Directive**

The `.END` directive terminates an assembly language program.

### **Syntax**

```
.END
```

### **Discussion**

This directive is the last statement in an assembly language program. If a source file lacks an `.END` directive, the Assembler terminates the program when it encounters the end of the file.

### **Example**

A file that omitted the last line of this sample program would produce identical results.

```
double      .CODE
            .EXPORT double,ENTRY
            .PROC
            .CALLINFO
            .ENTER
            ADD    %arg0,%arg0,%ret0
            .LEAVE
            .PROCEND
            .END
```

---

## **.ENDM Directive**

The `.ENDM` directive marks the end of a macro definition. The macro definition is entered into the macro table and the remaining source lines are read in and assembled. An `.ENDM` directive must always accompany a `.MACRO` directive.

### **Syntax**

```
.ENDM
```

### **Example**

This example defines the macro `QUADL`; it aligns the data specified in the macro parameters on quad word boundaries. The `.ENDM` directive delimits the end of the definition of `QUADL`.

```
QUADL .MACRO  WD1 , WD2 , WD3 , WD4  
      .ALIGN  16  
      .WORD  WD1  
      .ALIGN  16  
      .WORD  WD2  
      .ALIGN  16  
      .WORD  WD3  
      .ALIGN  16  
      .WORD  WD4  
      .ENDM
```



## **.ENTER and .LEAVE Pseudo-Operations**

The `.ENTER` and `.LEAVE` pseudo-operations mark a procedure's entry and exit points. They instruct the Assembler to generate procedure entry and exit code sequences based on the information provided in the `.CALLINFO` directive.

### **Syntax**

`.ENTER`

Lines of code

`.LEAVE`

### **Discussion**

The `.ENTER` pseudo-operation marks an entry point for the current procedure. Every procedure that follows the standard procedure-calling convention must contain one `.ENTER` pseudo-operation. The calling conventions are described in the documents under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>. The `.LEAVE` pseudo-operation marks a procedure's exit point. Every procedure that follows the procedure-calling convention must contain one `.LEAVE` pseudo-operation. See “.ENTRY and .EXIT Directives” on page 83 for exceptions.

When the Assembler encounters an `.ENTER` pseudo-operation, it generates an entry code sequence according to the parameters in the `.CALLINFO` directive for that procedure. Similarly, when the Assembler encounters a `.LEAVE` pseudo-operation, it generates an exit code sequence according to the parameters in the `.CALLINFO` directive for that procedure.

Assembler Directives and Pseudo-Operations  
**.ENTER and .LEAVE Pseudo-Operations**

## **Example**

This example shows the placement of the `.ENTER` and `.LEAVE` pseudo-operations.

```
        .SPACE $TEXT$  
        .SUBSPA $CODE$  
entrypt  
        .PROC  
        .CALLINFO  
        .ENTER  
        SHLADD      %arg0,%arg1,%ret0  
        .LEAVE  
        .PROCEND  
        .EXPORT entrypt,ENTRY  
        .END
```

---

## .ENTRY and .EXIT Directives

.ENTRY and .EXIT are compiler generated directives that mark the entry point and return point of the current procedure.

### Syntax

.ENTRY

Lines of Code

.EXIT

### Discussion

The .ENTRY directive signifies that the next instruction is the beginning of an entry point for the current procedure. The .EXIT directive signifies that the next instruction initiates a return from the current procedure. These directives must be used when .ENTER and .LEAVE are not present. .ENTRY and .EXIT are optional if the unwind region does not have a corresponding entry or exit. See the documents under the topic PA-RISC Architecture at URL: <http://www.software.hp.com/STK/>.

### Example

This example shows a sequence of compiler-generated assembly code.

```
.PROC
    .CALLINFO CALLER
    .ENTRY                                ; proc entry code follows
    STW      %r2,-20(%sp)                 ; stack the return pointer
    LDO      48(%sp),%sp                  ; set up user stack pointer
    ADDIL    L'$THISMODULE$-$global$,%r27 ; point to printf data
    .CALL
    BL       printf,2                     ; set up for printf call
    BL       printf,2                     ; call printf thru RP
    LDO      R'$THISMODULE$-$global$(%r1),%r26 ; insert argument to
printf L$exit1                             ; hide from linker
    LDW      -68(%sp),%r2                 ; get callee RP
    BV       0(%r2)                       ; exit thru RP
    .EXIT
    LDO      -48(%sp),%sp                 ; end of exit sequence
    .PROCEND                             ; delete stack frame
```

## .EQU Directive

The `.EQU` directive assigns an expression value to an identifier.

### Syntax

*symbolic\_name* `.EQU` value

### Parameters

*symbolic\_name* The name of the identifier to which the Assembler assigns the expression.

*value* An integer expression. The Assembler evaluates the expression, which must be absolute, and assigns this value to *symbolic\_name*. If the expression references other identifiers, each identifier must be defined before the `.EQU` directive attempts to evaluate the expression.

---

#### NOTE

The Assembler prohibits the use of relocatable symbols (instruction labels) and imported symbols as components of an `.EQU` expression.

---

### Example

This is a valid assembly program because the definition of `val1` comes before the definition of `val2`. Reversing the first two statements, however, produces an error condition.

```
val1 .EQU 0
val2 .EQU val1+4
      .SPACE $TEXT$
      .SUBSPA $CODE$
      LDW  val1,%r1
      STW  %r1,val2
      .END
```

---

## .EXPORT Directive

The `.EXPORT` directive allows symbols to be defined in one program and used in other programs.

### Syntax

```
.EXPORT symbol [ , type] [ , argument-description] ...
```

### Parameters

*symbol*      The name of an identifier whose definition is being exported or imported.

*type*        A linker symbol type that can take one of the following values:

ABSOLUTE	Designates an absolute symbol. In PA-RISC 2.0W (64-bit mode) ABSOLUTE symbols map to STT_NOTYPE with a section index of SHN_ABS.
DATA	Designates a data symbol. In PA-RISC 2.0W (64-bit mode) DATA symbols map to STT_OBJECT.
CODE	Designates a code location. The location can not be a procedure entry. In PA-RISC 2.0W (64-bit mode) CODE symbols map to STT_OBJECT.
ENTRY	Designates the entry point of a procedure. In PA-RISC 2.0W (64-bit mode) ENTRY symbols map to STT_FUNC.
MILLICODE	Locates code for the entry point of a millicode routine.

## Assembler Directives and Pseudo-Operations

### .EXPORT Directive

MILLI_EXT	Locates code for the entry point of an external millicode routine.
PLABEL	Locates a pointer to a procedure.
PRI_PROG	Designates the primary program entry point. The outer block of HP Pascal/HP-UX and the main program in HP FORTRAN 77/HP-UX are type PRI_PROG.  In PA-RISC 2.0W (64-bit mode) PRI_PROG symbols map to STT_FUNC.
SEC_PROG	Designates a secondary program entry point.  In PA-RISC 2.0W (64-bit mode) SEC_PROG symbols map to STT_FUNC.

#### *argument- description*

Allows you to communicate to the linker the types of registers used to receive floating point arguments and return floating point return results. Similarly, this information can be communicated in the .CALL directive.

The linker requires this information, since the Procedure Calling Convention described in the documents under the topic PA-RISC Architecture at <http://www.software.hp.com/STK/> allows floating point arguments and return values to reside in either general registers or floating point registers, depending on source language convention. At link time, the linker ensures that both the caller and called procedure agree on argument location. If not, the linker may insert code to relocate the arguments (or return result) before control is transferred to the called procedure or a procedure return is completed.

The form of *argument-description* is described in See “.CALL Directive” on page 63 in this chapter.

## Discussion

The `.EXPORT` directive uses a series of keywords to define a symbol to the linker. These keywords declare the symbol's type, and its argument relocation information if the symbol is the name of a procedure.

## Example

This example makes the symbol `proc` available to the linker as an entry point. It also specifies that the first argument is expected in a general register.

```
.EXPORT proc,ENTRY,ARGW0=GR
```

---

## **.FLOAT Pseudo-Operation**

The `.FLOAT` pseudo-operation initializes a single-word of storage to a floating-point value calculated from the parameters provided. If the location counter is not aligned on a word boundary, it is forced to the next multiple of four. If the statement is labeled, the label refers to the first byte of the storage area.

### **Syntax**

```
.FLOAT integer [ .fraction] [ E [ -] power]
```

```
.FLOAT .fraction [ E [ -] power]
```

### **Parameters**

<i>integer</i>	Specifies the whole number part of a decimal number.
<i>fraction</i>	Specifies the fractional part of a decimal number.
<i>power</i>	Specifies the power of ten to raise a decimal number. To raise the decimal number to a negative power of ten, place a minus sign (-) directly in front of the power specified.

### **Example**

Each of the following examples initializes one word of memory to floating-point quantities: 0.00096 and 3400000.0, respectively.

```
factor1 .FLOAT 9.6E-4
factor2 .FLOAT 3.4E6
```



## .HALF Pseudo-Operation

The `.HALF` pseudo-operation reserves storage and initializes it to the given value.

### Syntax

```
.HALF [ init_value [ , init_value ] ...]
```

### Parameters

*init\_value*      Either a decimal, octal, or hexadecimal number or a sequence of ASCII characters, surrounded by quotation marks. If you omit the initializing value, the Assembler initializes the area to zero.

### Discussion

The `.HALF` pseudo-operation requests 16 bits of storage. If the location counter is not properly aligned on a boundary for a data item of that size, the Assembler advances the location counter to the next multiple of that item's size before reserving the area.

When you label the pseudo-operation, the label refers to the first byte of the storage area. Operands separated by commas initialize successive units of storage.

### Example

This example allocates two half-words, initializing them to 50 and 100. The label `B` refers to the first half-word allocated.

```
B    .HALF    50,100
```

---

## **.IMPORT Directive**

The `.IMPORT` directive allows symbols to be defined in one program but used in other programs.

### **Syntax**

```
.IMPORT symbol [ , type] [ , TSPECIFIC]
```

### **Parameters**

*symbol*            The name of an identifier whose definition is being imported.

*type*                A linker symbol type that can take one of the following values:

ABSOLUTE	Designates an absolute symbol.
DATA	Designates a data symbol.
CODE	Designates a code location. The location can not be a procedure entry.
ENTRY	Designates the entry point of a procedure.
MILLICODE	Locates code for the entry point of a millicode routine.
MILLI_EXT	Locates code for the entry point of an external millicode routine.
PLABEL	Locates a pointer to a procedure.
PRI_PROG	Designates the primary program entry point. The outer block of HP Pascal/HP-UX and the main program in HP FORTRAN 77/HP-UX are type <code>PRI_PROG</code> .
SEC_PROG	Designates a secondary program entry point.

**TSPECIFIC**      The **TSPECIFIC** keyword indicates that this is a thread local storage symbol.

## Discussion

The `.IMPORT` directive uses a series of keywords to define a symbol to the linker. These keywords declare the symbol's type. Because the `.IMPORT` directive specifies that another object module contains this symbol's formal definition, the Assembler does not associate an imported symbol with any particular subspace. When an `.IMPORT` directive lacks a *type* parameter, the Assembler assigns the type of the current subspace (either `$CODE$` or `$DATA$`) to the symbol.

## Example

The `.IMPORT` directive lets the Assembler access `symname` as a recognized symbol, even though it is actually defined elsewhere. The linker resolves the difference.

```
.IMPORT symname, CODE           ;import symname as a CODE symbol.
.CODE                           ;begin CODE subspace
LDIL    L'symname,%r1
BLE,n   R'symname(%sr4,%r1)    ;call the procedure symname in %sr4
space.
NOP
.END
```

---

## **.LABEL Directive**

The `.LABEL` directive permits a label definition to appear within a sequence of instructions that occur on a single line.

### **Syntax**

`.LABEL label_id`

### **Parameters**

*label\_id*            Names the label identifier.

---

**NOTE**

The `.LABEL` directive is especially useful when using the M4 macro processor or the C preprocessor (`cpp`). You would normally use this directive in a `DEFINE` macro that includes multiple instructions.

---

### **Example**

This example defines a `cpp` macro named `Loop`.

```
#define Loop(xx) LDO xx(%r0),%r1 ! .LABEL Loop ! ADDI,=-1,%r1,%r1 \  
! BL Loop,%r0 ! NOP ! LDI 1,%ret0 ; macro  
    .CODE  
step_ten  
    .PROC  
    .CALLINFO  
    .ENTER  
    Loop(10)  
    .LEAVE  
    .PROCEND  
    .EXPORT step_ten,ENTRY
```

---

## .LEVEL Directive

The `.LEVEL` directive tells the Assembler which version level of the PA-RISC architecture to accept while assembling the source file. The `.LEVEL` directive also tells the Assembler which implementations-specific features are used in the assembly source file.

### Syntax

$$\text{.LEVEL} \left\{ \begin{array}{l} 1.0 \\ 1.1 \\ 2.0 \\ 2.0W \end{array} \right\}$$

### Parameters

1.0	Enables PA-RISC 1.0 features. This is the default.
1.1	Enables PA-RISC 1.1 features.
2.0	Enables PA-RISC 2.0 features.
2.0W	Enables PA-RISC 2.0W features and assembles the source for a 64-bit machine.

### Discussion

The Assembler marks the relocatable object file to indicate the minimum PA-RISC architecture version level required when executing the object code corresponding to the source file. The linker marks the program file with the highest version level required by any of the object files linked into the program.

The Assembler uses the `.LEVEL` directive to mark the relocatable object file with the proper PA-RISC architecture version level. For example, if the code is expected to run only on PA-RISC 1.1 architectures, a `.LEVEL 1.1` should be inserted at the beginning of the source file.

To assemble a source file for a PA-RISC 64-bit system, use a `.LEVEL 2.0W` directive as the first directive in the source file.

### **.LEVEL Directive**

In the source file, the Assembler emits warning messages whenever a feature is used that is not appropriate for the specified `.LEVEL` directive. The default is to produce a PA-RISC 1.0 relocatable object file. If the default is used, any use of PA-RISC 1.1 or 2.0 features in the assembly source file generates a warning messages.

If the code is expected to run on more than one level of PA-RISC architecture, a run-time check should be used with a `.ALLOW` directive. See “`.ALLOW Directive`” on page 58 in this chapter for an example of a run-time check.

The `.LEVEL` directive is also used to indicate any implementation-specific extensions that the source file depends on. The Assembler marks the relocatable object file with information that indicates any implementation-specific extensions that were specified in the `.LEVEL` directive. The default for an assembly source file is no implementation-specific extensions; the Assembler generates warning messages if an implementation-specific extension is used.

---

## .LISTOFF and .LISTON Directives

The .LISTOFF and .LISTON directives control the expansion of instructions for all macro invocations, all predefined subspace declarations, and the .ENTER and .LEAVE pseudo-operations. .LISTOFF causes the Assembler to cease listing expanded instructions until a .LISTON directive is encountered. .LISTON causes the Assembler to list expanded instructions until a .LISTOFF directive is encountered.

The default is .LISTON.

### Syntax

.LISTOFF

.LISTON

### Example

The following is the definition of the macro `DECR`. It is referred to in the assembly listing generated when `.LISTON` was used with a procedure containing the macro invocation.

```
DECR      .MACRO          LAB,VAL
SKF       ADDIL          L'VAL-$global$,%dp
          LDW            R'VAL-$global$(%r1),%r20
LAB       ADDIBF,=,N     -1,%r20,LAB
          .ENDM

          .CODE
          .IMPORT $global$
          .IMPORT mark
          .IMPORT count
          .PROC
call_DECR .CALLINFO     FRAME=0, SAVE_RP
          .ENTER
          DECR          mark,count
          .LEAVE
          .PROCEND
```

## Assembler Directives and Pseudo-Operations

### .LISTOFF and .LISTON Directives

If `.LISTOFF` had been used in the above example, the macro invocation `DECR`, and the directives `.CODE`, `.DATA`, `.ENTER`, and `.LEAVE`, would not have been expanded in the assembly listing.

```
line offset  hexcode  label      opcode      operands (comment)
1          .LISTON
2          .CODE
          .SPACE $TEXT$,      SPNUM=0, SORT=0
          .SUBSPA $CODE$,     QUAD=0, ALIGN=8, ACCESS=0x2c
          .PROC
3
4          call_DECR          ;proc label
5          .CALLINFO         FRAME=0, SAVE_RP
6          .ENTER
          00000000 (6BC23FD9)      STW          2, -0x14(0, 0x1E)
          00000004 (37DE0060)      LDO          0x30(0x1E), 0x1E
7          00000008 (2B600000)      ADDIL       L'count-$global$, %dp
8          0000000C (683A0000)      STW          %arg0, R'count-$global$(%r1)
9          DECR              mark, count;
          macro invocation
          00000010 (2B600000)      ADDIL       L'VAL-$global$, %dp
          00000014 (48340000)      LDW         R'VAL-$global$(%r1), %r20
          LAB
          00000018 (AE9F3FF5)      ADDIBF, =   -1, %r20, LAB
          0000001C (08000240)      NOP
10         .LEAVE
          00000020 (4BC23F79)      LDW         -0x44(0, 0x1E), 2
          00000024 (E840C000)      BV          0(2)
          00000028 (37C03FA1)      LDO         -0x30(0x1E), 0
11         .PROCEND
12         .EXPORT          call_DECR, ENTRY
13         DATA
          .SPACE $PRIVATE$,   SPNUM=1, SORT=16
          .SUBSPA $DATA$,     QUAD=1, ALIGN=8
          ACCESS=0x1f
14         .IMPORT          $global$
15         40000000 (00000000) count .WORD      0
16         .LISTOFF
```



---

## .LOCCT Directive

The `.LOCCT` directive specifies where subsequent code should occur in one of the four location counters of the current subspace.

### Syntax

```
.LOCCT [ loc_number]
```

### Parameters

*loc\_number*      A location-counter number of the current subspace. The permissible values are 0, 1, 2, and 3. The default is zero.

---

#### NOTE

The `.LOCCT` directive is not permitted within a procedure and cannot be used to produce unwindable code.

---

### Example

This example uses two location counters to separate code from data. In the assembled code, everything under location counter 0 comes first, followed by everything under location counter 1, and so on.

```
        .CODE
        .LOCCT 0
ldvall  LDIL    L'vall,%r19
        LDO    R'vall(%r19),%r19
        .LOCCT 1
vall    .WORD   57005
        .LOCCT 0
ldval2  LDIL    L'val2,%r20
        LDO    R'val2(%r20),%r20
        .LOCCT 1
val2    .WORD   61453
```

## .MACRO Directive

The `.MACRO` directive marks the beginning of a macro definition. An `.ENDM` directive must be used to end the macro definition.

### Syntax

```
label .MACRO [ formal_parameter{,formal_parameter}...]
```

### Parameters

*label*

The name of the macro.

*formal\_parameter*

Specifies a string of characters treated as a positional parameter. The *i*th actual parameter in a macro invocation is substituted for the *i*th formal parameter in the macro declaration wherever the formal parameter appears in the body of the macro definition.

### Discussion

Normal Assembler syntax is observed within macro definitions, except that text substitution is assumed for formal parameters. The following line is an example of a macro declaration:

```
DECRA .MACRO LAB,VAL
```

`LAB` and `VAL` are formal parameters. Their actual values are determined by the first and second parameters on any invocation of the macro `DECRA`. On the macro invocation, the parameters are delimited by commas. Successive commas indicate a null parameter, causing the expanded macro to substitute null for one of its formal parameters. When the number of formal parameters exceeds the number of actual parameters, null parameters are inserted for the excess parameter positions. When the number of actual parameters exceeds the number of formal parameters, a warning is issued and the excess parameters are ignored.

---

**NOTE** Although there is no upper limit on the number of parameters or arguments in a macro definition, no single macro parameter may exceed 200 characters.

---

Macro definitions may appear wherever and as often as necessary within source code. Macro definitions may occur inside or outside of spaces, subspaces, and procedures.

Because the Assembler always uses the most recently encountered definition, macros may be redefined as often as desired.

---

**NOTE** A macro cannot be defined within the body of another macro definition.

---

Although nested macro definitions are not allowed, nested macro calls are. A nested macro call occurs when one macro is invoked within the definition of another macro. A macro may not be invoked within its own definition. Macros can only be invoked after being defined.

## Examples

The macro definition defines a simple counter or timer called `DECR`.

```
DECR      .MACRO      LAB,VAL
SETP     ADDIL      L'VAL-$$global$,%dp
          LDW       R'VAL-$$global$(%r1),%r20
LAB
          ADDIBF,=  -1,%r20,LAB
          NOP
          .ENDM
```

The following is an invocation of `DECR`:

```
DECR LOOP,COUNT
```

`LOOP` and `COUNT` are the actual parameters that are specific to this particular invocation of the macro `DECR`.

During macro expansion, textual substitution for positional parameters is performed in the body of the macro definition. Substitution is performed on strings of characters that are delimited by blanks, tabs, commas, or semicolons. If the string matches one of the formal parameters, it is replaced with the corresponding actual parameter.

When a macro definition contains a label, the expanded form of the macro adds a unique suffix to the label for each instance the macro is invoked. This unique suffix prevents duplicate symbols from occurring

**.MACRO Directive**

and prevents the label from being referenced from outside the body of the macro definition. This suffix also contains a number that is used as a counter by the Assembler.

The following example defines the macro PRINT, which calls the printf() function (see *printf(3S)* in *HP-UX Reference*). The macro parameter DATA\_ADDR is used to set up the argument to be passed to printf().

```
PRINT    .MACRO    DATA_ADDR
         ADDIL    L'DATA_ADDR,%dp
         .CALL
         BL      printf,%rp
         LDO     R'DATA_ADDR(%r1),%arg0
         .ENDM
```

The next example defines the macro STORE. STORE places the contents of the register REG, the first macro parameter, into the memory address LOC, the second parameter.

```
STORE    .MACRO    REG,LOC
         LDIL    L'LOC-$global$,%r1
         STW    REG,R'LOC-$global$(%r1)
         .ENDM
```

---

## .ORIGIN Directive

The `.ORIGIN` directive advances the location counter to the specified location.

### Syntax

```
.ORIGIN [ location]
```

### Parameters

*location*      The integer value used to advance the location counter to that absolute location. The location counter value may not decrease during this process; that is, the value specified cannot be less than the value of the current location counter.

The default value is zero.

### Discussion

When the Assembler encounters an `.ORIGIN` directive, it issues a `.BLOCK` pseudo-operation of a size calculated to advance the location counter to the requested origin. See “`.BLOCK` and `.BLOCKZ` Pseudo-Operations” on page 60 in this chapter.

### Example

This sample program performs an exclusive OR, advances the location counter to 64 bytes, and branches to the label `idx`.

```
        .CODE
        XOR    %r21,%r22,%r23
        B     idx
        NOP
        .ORIGIN 64
idx     LDWX   %r23(%sr0,%sr0),%r3
        .END
```

## **.PROC and .PROCEND Directives**

The `.PROC` and `.PROCEND` directives bracket the instructions within a procedure.

### **Syntax**

`.PROC`

Lines of Code

`.PROCEND`

### **Discussion**

The `.PROC` directive signifies that the next instruction is the first instruction of a procedure. The `.PROCEND` directive signifies that the previous instruction was the last instruction of the procedure. Switching spaces or subspaces within a procedure is not permitted.

Every procedure must contain a `.CALLINFO` directive and normally contains an `.ENTER` and `.LEAVE` pseudo-operation. The only exception to the latter rule occurs in procedures that are either compiler-generated or created by programmers who are writing their own entry and exit code sequences. In this case, you must use the `.ENTRY` and `.EXIT` compiler directives.

---

#### **NOTE**

Because the `.ENTER` and `.LEAVE` pseudo-operations guarantee that the stack unwind process works correctly, you should consistently use these directives rather than writing your own entry and exit code sequences.

---

## Example

This template shows a procedure that follows the procedure-calling convention.

```
.CODE
test
  .PROC
  .CALLINFO
  .ENTER
  COMCLR,=    %arg0,%arg1,%ret0
  LDI        1,%ret0
  .LEAVE
  .PROCEND
  .EXPORT test
```

## **.REG Directive**

The `.REG` directive assigns a predefined or user-defined typed-register to a symbol, which becomes a synonym for the typed-register.

### **Syntax**

```
label .REG [ typed_register]
```

### **Parameters**

*label*                    A user-defined register name.

*typed\_register*        Must be one of the predefined Assembler registers or a previously defined user-defined register name All predefined Assembler registers begin with `%`.

### **Example**

This example defines the register `shift` as a synonym for control register eleven. `%sar` is a predefined synonym for control register eleven, the shift-amount register.

```
shift    .REG %sar
```



## **.SHLIB\_VERSION Directive**

The `.SHLIB_VERSION` directive marks the object file with a version string that the shared library understands.

### **Syntax**

```
.SHLIB_VERSION " mm/yyyy"
```

### **Parameters**

*mm*                   The one- or two-digit number of the month.  
*yyyy*                   The four-digit number of the year.

### **Example**

The following pseudo-operation places the date September 1994 in the object file.

```
.SHLIB_VERSION "9/1994"
```

---

## .SPACE Directive

The `.SPACE` directive starts a new space or switches back to an old space. The Assembler ignores the `.SPACE` directive for 64-bit assembly programs. For more information, see “Sections in 64-bit Mode” on page 44.

### Syntax

```
.SPACE name [ ,NOTDEFINED
               ,PRIVATE
               ,SORT=value
               ,SPNUM=value ...
               ,TSPECIFIC
               ,UNLOADABLE
```

### Parameters

<i>name</i>	An identifier that names the new space.
NOTDEFINED	Specifies that the definition for this space occurs in another object module.
PRIVATE	Specifies that other programs cannot <i>share</i> the data in this space. The enforcement of this directive depends on the operating system.
SORT= <i>value</i>	Provides an integer value for the sort key. The linker orders the spaces in the output object module according to this key. It is suggested that the number “8” be used for space <code>\$TEXT\$</code> and the number “16” be used for <code>\$PRIVATE\$</code> .
SPNUM= <i>value</i>	A space number constant that provides a specific number for the current space. Its use is currently optional and is ignored by the linker. If the first parameter of the <code>.SPACE</code> directive is an integer, it will be interpreted as the space number and any remaining parameters will be ignored.

TSPECIFIC	Indicates that this space contains thread local storage data.
UNLOADABLE	Specifies that the space resides on disk and is not loadable into main memory. Debugger data is a typical example of an unloadable space.

## Discussion

The first time the Assembler encounters a `.SPACE` directive with a new *name*, it uses that name to declare a new space. As this is the defining occurrence of that space, additional keywords can describe attributes for that space.

If the Assembler encounters subsequent `.SPACE` directives with that *name*, it continues that space. In this case, where the program is re-entering a previously defined space, the `.SPACE` directive can only contain the space name; other keywords to describe the space are illegal.

A space can contain from one to four discrete quadrants (See the `QUAD` parameter of the `.SUBSPA` directive.) When you divide a space into multiple quadrants, you must define all the subspaces within each quadrant as a group. If subspaces for a quadrant are defined individually, program operation is unpredictable. The Assembler, however, does not check for this condition.

## Example

This example shows some of the standard “space” definitions in a typical assembly language program.

```
.SPACE $TEXT$, SPNUM=0, SORT=8
.SPACE $PRIVATE$, SPNUM=1, PRIVATE, SORT=16
.SPACE $myspace$, SPNUM=7, UNLOADABLE
.SPACE $THREAD_SPECIFIC$, PRIVATE, TSPECIFIC, SORT=32
```

---

## **.SPNUM Pseudo-Operation**

The `.SPNUM` pseudo-operation reserves a word of storage and initializes it with the space number of the space named by the operand. Only one operand is allowed and any label present is offset at the first byte of the storage just initialized.

### **Syntax**

`.SPNUM name`

### **Parameters**

*name* Specifies the name of a space whose space number is used to initialize a word of storage.

---

#### **NOTE**

Space numbers are ignored by the linker.

### **Example**

In this example, the space number of `$PRIVATE$`, 1, is stored as the address of the symbol `LOG` by the `.SPNUM` pseudo-operation.

```

                .SPACE    $PRIVATE$,SPNUM=1    SORT=16
                .SUBSPA   $DATA$,QUAD=1,      ALIGN=8,ACCESS=0x1f    SORT=24
data_ref
                .WORD
LOG             .SPNUM    0xFFFF
                .SPNUM    $PRIVATE$

```

## **.STRING and .STRINGZ Pseudo-Operations**

The `.STRING` pseudo-operation reserves storage for a data area and initializes it to ASCII values. The `.STRINGZ` pseudo-operation reserves storage the same as `.STRING`, but appends a zero byte to the data. This creates a C-language-type string. If the statement is labeled, the label refers to the first byte of the storage area.

### **Syntax**

```
.STRING "init_value"  
.STRINGZ "init_value"
```

### **Parameters**

*init\_value*      A sequence of ASCII characters, surrounded by quotation marks. A string can contain up to 256 characters. The enclosing quotation marks are not stored.

The following escape sequences can be used to represent characters:

<code>\"</code>	Quotation mark
<code>\0</code>	Null ( $=\backslash x00$ ; ASCII NUL)
<code>\\</code>	Backslash
<code>\b</code>	Backspace ( $=\backslash x08$ ; ASCII BS)
<code>\f</code>	Form feed ( $=\backslash x0C$ ; ASCII FF)
<code>\n</code>	Newline ( $=\backslash x0A$ ; ASCII LF)
<code>\r</code>	Carriage return ( $=\backslash x0D$ ; ASCII CR)
<code>\t</code>	Tab ( $=\backslash x09$ ; ASCII HT)
<code>\xhh</code> or <code>\xhh</code>	Any 8-bit character; <i>hh</i> is two hexadecimal digits.

## Discussion

The `.STRING` pseudo-operation requests the required number of bytes to store the string (where each character is stored in a byte). The `.STRINGZ` pseudo-operation also requests the required storage for the quoted string but then appends a zero byte for compatibility with C language strings.

When you label one of these pseudo-operations, the label refers to the first byte of the storage area.

## Examples

This pseudo-operation allocates eight bytes, the first of which is labeled `G`. Then it initializes this area with the characters: `A`, *space*, `S`, `T`, `R`, `I`, `N`, and `G`.

```
G .STRING "A STRING"
```

This pseudo-operation allocates eight bytes to hold `A STRING`, allocates an additional byte for the appended zero, and associates the label `G` with the first byte of the storage area.

```
G .STRINGZ "A STRING"
```

---

## .SUBSPA Directive

The `.SUBSPA` directive declares a new subspace or switches back to an old subspace. In 64-bit assembly programs subspaces map directly to the ELF concept of sections, so the `.SUBSPA` directive switches to or creates a new section. For more information, see “Sections in 64-bit Mode” on page 44.

### Syntax

```
.SUBSPA name [,ACCESS=value]  
              [,ALIGN=value]  
              [,CODE_ONLY  
              [,COMMON  
              [,DUP_COMM  
              [,FIRST  
              [,FROZEN ...  
              [,LOCK  
              [,QUAD=value]  
              [,SORT=value]  
              [,TSPECIFIC  
              [,UNLOADABLE  
              [,ZERO
```

### Parameters

- |                                  |   |
|----------------------------------|---|
| <i>name</i>                      | An identifier that names the current subspace.  |
| <code>ACCESS=<i>value</i></code> | Specifies the 7-bit value for the access rights field in the <code>PDIR</code> (Physical Page Directory for virtual address mapping). Must be <code>0X2C</code> for code, and <code>0X1F</code> for data subspaces.                 |
| <code>ALIGN=<i>value</i></code>  | Specifies a value (which must be a power of 2) on which the Assembler should align the beginning of the subspace. The default value is the largest alignment requested within that subspace, or one if no alignment requests exist. |

**.SUBSPA Directive**

CODE_ONLY	Specifies that this subspace contains only code.
COMMON	Specifies that this subspace is a common block.
DUP_COMM	Specifies that the initialized data symbols within this subspace can have duplicate names. When you include this parameter, multiple occurrences of a universal data symbol can exist and the linker does not report a “Duplicate Definition” error.
FIRST	Specifies that the subspace must be allocated exactly at the beginning of the specified space.
FROZEN	If set, the subspace is locked into physical memory when the subspace goes into execution.
LOCK	If set, the subspace is locked into physical memory when the operating system is booted.
QUAD= <i>value</i>	Specifies the quadrant (0 through 3) in which the Assembler should place this subspace. The default value is zero.
SORT= <i>value</i>	Provides an integer value for the primary sort key. The linker orders the subspaces in the output object module according to this primary key. If several subspaces share the same primary key value, the linker lists these subspaces in the order in which it processes them. It is suggested that 24 be used for both code and data subspaces.
TSPECIFIC	Indicates that this space contains thread local storage data.
UNLOADABLE	Specifies that this subspace is not loadable into memory. Loadable subspaces must reside in loadable spaces, and unloadable subspaces must reside in unloadable spaces.
ZERO	Specifies that this subspace contains all zeros and no data appears in the output file.



## Discussion

The first time the Assembler encounters a `.SUBSPA` directive with a new *name*, it uses that name to declare a new subspace. As this is the defining occurrence of that subspace, optional keywords describe attributes of that subspace.

When the Assembler encounters additional `.SUBSPA` directives with that *name*, it continues that subspace. In this case, the `.SUBSPA` directive can only contain the subspace name; other keywords to describe the subspace are illegal.

## Example

This example shows some of the standard “subspace” definitions in a typical assembly language program.

```
.SUBSPA $CODE$,    QUAD=0,ALIGN=8,ACCESS=0x2c, SORT=24, CODE_ONLY
.SUBSPA $DATA$,    QUAD=1,ALIGN=8,ACCESS=0x1f, SORT=16
.SUBSPA $TBSS$,    QUAD=1,ALIGN=8,ACCESS=0x1f,ZERO, TSPECIFIC, SORT=40
```

---

## **.VERSION Directive**

The `.VERSION` directive places the designated string in the current object module for version identification.

### **Syntax**

```
.VERSION "info_string"
```

### **Parameters**

*info\_string*      A sequence of ASCII characters, surrounded by quotation marks. The string can contain up to 256 characters.

### **Discussion**

The Assembler places this string in the current object module. A program can contain multiple `.VERSION` directives.

Once the version information is in the object file, the HP-UX utility `strings` can be used to access it. See *strings(1)* in *HP-UX Reference*.

---

**NOTE**

---

This directive can appear anywhere in the source file, and multiple occurrences are permitted.

### **Example**

This program inserts version information into the object module, and performs subtract and deposit operations.

```
.CODE
.VERSION "Version 1 of This Simple Sample Program"
SUB     %r19,%r20,%r19
DEP     %r19,14,5,%r22
.END
```

---

## .WORD Pseudo-Operation

The `.WORD` pseudo-operation reserves storage and initializes it to the given value.

### Syntax

```
.WORD [ init_value [ , init_value ] ...]
```

### Parameters

*init\_value*      A relocatable or absolute expression, a decimal, octal, or hexadecimal number, or a sequence of ASCII characters surrounded by quotation marks. If you omit the initializing value, the Assembler initializes the area to zero.

### Discussion

The `.WORD` pseudo-operation requests 32 bits of storage. If the location counter is not properly aligned on a boundary for a data item of that size, the Assembler advances the location counter to the next multiple of that item's size before reserving the area.

When you label the pseudo-operation, the label refers to the first byte of the storage area. Operands separated by commas initialize successive units of storage.

### Example

The first pseudo-operation advances the current subspace's location counter to a word boundary, allocates a word of storage labeled `F` and initializes that word to minus 32 (2's complement). The second pseudo-operation initializes a word of storage to the hexadecimal number `6EFF1234`.

```
F     .WORD     -32  
       .WORD     0X6eff1234
```

## Programming Aids

The Assembler provides a series of standard space and subspace definitions that you can use to simplify the writing of an assembly program. These definitions are duplicated in the system file `/usr/lib/pcc_prefix.s`. Because this file is relatively large and may change with new releases of the Assembler, you can view the most recent version of the file on your terminal screen by typing the command:

```
more /usr/lib/pcc_prefix.s
```

Table 4-3 on page 116 lists the predefined directives for establishing standard spaces and subspaces.

**Table 4-3 Predefined Subspace Directives**

Directive	Space Name	Default Parameters
.BSS	.space \$PRIVATE\$, ' .subspa \$BSS\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=82, ZERO
.CODE	.space \$TEXT\$, ' .subspa \$CODE\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x2c, SORT=24
.DATA	.space \$PRIVATE\$, ' .subspa \$DATA\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=24
.FIRST	.space \$TEXT\$, ' .subspa \$FIRST\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=2048, ACCESS=0x2c, SORT=4, FIRST
.GATE	.space \$TEXT\$, ' .subspa \$GATE\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x4c, SORT=84, CODE_ONLY
.GLOBAL	.space \$PRIVATE\$, ' .subspa \$GLOBAL\$, .IMPORT \$global\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=40
.GNTT	.space \$DEBUG\$, ' .subspa \$GNTT\$,	SPNUM=2, PRIVATE, UNLOADABLE, SORT=80 ALIGN=4, ACCESS=0, UNLOADABLE
.HEADER	.space \$DEBUG\$, ' .subspa \$HEADER\$,	SPNUM=2, PRIVATE, UNLOADABLE, SORT=80 ALIGN=4, ACCESS=0, UNLOADABLE, FIRST
.HEAP	.space \$PRIVATE\$, ' .subspa \$HEAP\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=82
.LIT	.space \$TEXT\$, ' .subspa \$LIT\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x2c, SORT=16

<b>Directive</b>	<b>Space Name</b>	<b>Default Parameters</b>
.LNTT	.space \$DEBUG\$, ' .subspa \$LNTT\$,	SPNUM=2, PRIVATE, UNLOADABLE, SORT=80 ALIGN=4, ACCESS=0, UNLOADABLE
.MILLICODE	.space \$TEXT\$, ' .subspa \$MILLICODE\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x2c, SORT=8
.PCB	.space \$PRIVATE\$, ' .subspa \$PCB\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x10, SORT=82
.REAL	.space \$TEXT\$, ' .subspa \$REAL\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x2c, SORT=4, FIRST, LOCK
.RECOVER	.space \$TEXT\$, ' .subspa \$RECOVER\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=4, ACCESS=0x2c, SORT=80
.RESERVED	.space \$TEXT\$, ' .subspa \$RESERVED\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=8, ACCESS=0x73, SORT=82
.SHORTDATA	.space \$PRIVATE\$, ' .subspa \$SHORTDATA\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=16
.SLT	.space \$DEBUG\$, ' .subspa \$SLT\$,	SPNUM=2, PRIVATE, UNLOADABLE, SORT=80 ALIGN=4, ACCESS=0, UNLOADABLE
.STACK	.space \$PRIVATE\$, ' .subspa \$STACK\$,	PRIVATE, SPNUM=1, SORT=16 QUAD=1, ALIGN=8, ACCESS=0x1f, SORT=82
.UNWIND	.space \$TEXT\$, ' .subspa \$UNWIND\$,	SPNUM=0, SORT=8 QUAD=0, ALIGN=4, ACCESS=0x2c, SORT=64
.VT	.space \$DEBUG\$, ' .subspa \$VT\$,	SPNUM=2, PRIVATE, UNLOADABLE, SORT=80 ALIGN=4, ACCESS=0, UNLOADABLE

Assembler Directives and Pseudo-Operations  
**Programming Aids**

---

## 5

# Pseudo-Instruction Set

In addition to the PA-RISC instruction set, which is described in *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and *PA-RISC 2.0 Architecture*, the Assembler provides a number of pseudo-instructions that perform commonly used forms of the basic instructions. These pseudo-instructions are listed with their standard-instruction format in Table 5-1 on page 120.

---

### NOTE

When coding branch instructions, including those with nullification specified, pay attention to the instruction that follows the branch in the source code. All branch instructions consider this *following* instruction to be in their *delay slot*. You can use a `NOB` pseudo-instruction to fill the delay slot when there is no other useful work to be performed. This delay slot is usually executed.

---

**Table 5-1 Pseudo-Instructions**

<b>Pseudo-Instruction Format</b>	<b>Standard Instruction Format</b>
ADDB, cond, n <sup>1, 2</sup> r1, r2, target	ADDBT, cond, n      r1, r2, target ADDBF, cond, n      r1, r2, target
ADDIB, cond, n <sup>1, 2</sup> r1, r2, target	ADDIBT, cond, n      r1, r2, target ADDIBF, cond, n      r1, r2, target
B                              w	BL                      w, %r0
COMB, cond, n <sup>1, 2</sup> r1, r2, target	COMBF, cond, n      r1, r2, target COMBT, cond, n      r1, r2, target
COMIB, cond, n <sup>1, 2</sup> r1, r2, target	COMIBF, cond, n      r1, r2, target COMIBT, cond, n      r1, r2, target
COPY                      r, t	LDO                      0(r), t
LDI                          i, t	LDO                      i(%r0), t
MTSAR                      r	MTCTL                  r, %cr11
NOP	OR                      %r0, %r0, %r0

1. The `cond` completer determines the actual instruction that the Assembler uses in the conditional branch. The `T` form is used with nonnegated completers. The `F` form is used with negated completers. See Table 5-2 on page 121 and Table 5-3 on page 122 for details.
2. `n` indicates an optional nullification completer.



**Table 5-2 Compare and Branch Conditions (COMB and COMIB)**

<b>cond</b>	<b>Description</b>	
	<b>never</b>	
=	opd1 is equal to opd2	Nonnegated
<	opd1 is less than opd2 (signed)	
<=	opd1 is less than or equal to opd2 (signed)	
<<	opd1 is less than opd2 (unsigned)	
<<=	opd1 is less than or equal to opd2 (unsigned)	
SV	opd1 minus opd2 results in overflow (signed)	
OD	result of opd1 minus opd2 is odd	
TR	<b>always</b>	Negated
<>	opd1 is less than or greater than opd2	
>=	opd1 is greater than or equal to opd2 (signed)	
>	opd1 is greater than opd2 (signed)	
>>=	opd1 is greater than or equal to opd2 (unsigned)	
>>	opd1 is greater than opd2 (unsigned)	
NSV	opd1 minus opd2 results in no overflow (signed)	
EV	result of opd1 minus opd2 is even	

**Table 5-3 Add and Branch Conditions (ADDB and ADDIB)**

<b>cond</b>	<b>Description</b>	
	<b>never</b>	
=	opd1 is equal to -opd2	Nonnegated
<	opd1 is less than -opd2 (signed)	
<=	opd1 is less than or equal to -opd2 (signed)	
NUV	opd1 + opd2 < $2^{32}$ (no unsigned overflow)	
ZNV	opd1 + opd2 < $2^{32}$ or opd1 + opd2 = 0	
SV	opd1 plus opd2 results in overflow (signed)	
OD	result of opd1 plus opd2 is odd	
	<b>always</b>	
<>	opd1 is not equal to -opd2	Negated
>=	opd1 is greater than or equal to -opd2 (signed)	
>	opd1 is greater than -opd2 (signed)	
UV	opd1 + opd2 > = $2^{32}$ (unsigned overflow)	
VNZ	opd1 + opd2 > $2^{32}$ and opd1 + opd2 not = 0	
NSV	opd1 plus opd2 results in no overflow (signed)	
EV	result of opd1 plus opd2 is even	

---

## 6

# Assembling Your Program

This chapter describes two different ways you can invoke the Assembler and the various command line options controlling its behavior. It also contains a brief description of the interface between the Assembler and linker, and things you should remember to facilitate the running of an assembly program.

---

## Invoking the Assembler

You can invoke the Assembler directly by using the `as` command. Or, you can invoke the Assembler through the `cc` command, which processes the assembly source using the C preprocessor. The next two sections describe these pathways.

## Using the `as` Command

The `as` command is the standard command for invoking the Assembler on PA-RISC systems running on HP-UX. See `as(1)` in *HP-UX Reference* for complete details.

If no files are specified, the Assembler reads source text from standard input, which must be a command-line pipe or a FIFO. It cannot be a device file, such as a terminal.

The Assembler produces a single output file (see the `-o` option). If the source text is read from standard input, the object file is written to standard output and the `-o` option is ignored.

The `as` command resides in the `/usr/ccs/bin` directory.

If your programming environment does not establish a path to this directory, you must include the path name as the first part of the `as` command. For example:

```
/usr/ccs/bin/as -l line.s box.s draw.s
```

### Syntax

```
as [ [ option] ...[ file] ...] ...
```

### Parameters

*file*                   The name of an input file. The name must include the suffix `.s`. If you specify multiple input files, they are concatenated in order.

*option*                A flag telling the Assembler to take some special action. All options affect all input files. The `as` command supports the following options.

`+DAarch.`            Assemble code for the architecture

specified. The use of this option is discouraged. The preferred method for selecting the architecture is to use a `.LEVEL` directive in the assembly source file.

The target architecture specified with the `.LEVEL` directive takes precedence over the architecture specified with the `+DA` option.

- `-e` Specify that the Assembler should tolerate one million errors before terminating the assembly process. Without this option, the Assembler terminates a program after 100 errors.
- `-f` Specify that procedures call other procedures as the default condition. Normally, the Assembler assumes that procedures do not call other procedures. (See the `CALLS` and `NO_CALLS` parameters for `.CALLINFO` directive.).
- `-l` List the assembled program on standard output. This listing shows instruction offsets and the values stored in each field.
- `-o filename` Assign the specified name to the output file. The default name for the output file is the name of the last input file with the suffix changed to `.o`.
- `-p level` Specify the privilege level of running capability. *level* must be in the range zero to three, with three being the least privileged (normal user).
- `-s` Set the output file suffix to `.ss` instead of `.o`. The file will have a format suitable for conversion to the ROM burning programs.

Assembling Your Program  
Using the `as` Command

- `-u` Prevent the Assembler from creating stack unwind descriptors. This option precludes the use of the `.ENTER` and `.LEAVE` directives within a program.
  - `-v filename` Name a file to which the Assembler writes cross-reference information; this includes the source file and the line number for each appearance of all symbols.
  - `-V` Print the version number of the Assembler program to standard error before assembling the source text.
  - `-w[number]` Either suppress all warning messages if no number is supplied or suppress just the warning number provided. You can use multiple `-wnumber` options to suppress additional warning messages.
  - `+z` Create position-independent code suitable for inclusion in a shared library. Use this option for small linkage tables.
  - `+Z` Create position-independent code suitable for inclusion in a shared library. Use this option for large linkage tables.
- For more information about linkage tables, see *HP-UX Linker and Libraries Online User Guide*.

---

## Using the cc Command

You can also use the `cc` command to run the Assembler on files that have a `.s` suffix. See `cc(1)` man page for the HP C/HP-UX ANSI C compiler, if installed. The `cc` command inserts the system file

```
/usr/lib/pcc_prefix.s
```

in front of the `.s` file and pipes the file through the C preprocessor (see `cpp(1)` in *HP-UX Reference*) before passing the file to the Assembler. `pcc_prefix.s` is a concatenation of the following header files in the directory `/usr/include`:

<code>hard_reg.h</code>	Set of <code>.REGs</code> for hardware registers.
<code>soft_reg.h</code>	Set of register definitions that follows the Procedure Calling Convention.
<code>std_space.h</code>	Set of space and subspace definitions that most Assembler programs use.

---

### NOTE

If you are using the HP C/HP-UX ANSI C compiler, you can suppress the `pcc_prefix.s` file with the `cc` option `+a`.

---

## Passing Arguments to the Assembler

The `cc` command normally strips all `as` command options from the command line, writing a warning to standard error. Therefore, when you want to retain one of these options, you must include the `-Wa` command-line option

```
-Wa, ,as-argument [ as-argument] ...
```

*as-argument* names an Assembler argument you want to preserve. For example, to specify a cross-reference file, you could use:

```
-Wa, -v, myxreffile
```

Similarly, you can pass options to the C preprocessor (`cpp`) or the linker (`ld`) with `-Wp` and `-Wl`, respectively.

## **cpp Preprocessor**

You can use the C preprocessor (`cpp`) with assembly language programs to include C-type macros, including directives. You can use an exclamation point (!) as a statement terminator to include multiple statements in the body of one macro definition. Furthermore, you can use the `.LABEL` directive to declare labels within a macro definition.

---

**NOTE**

---

If you use `cpp`, C-style comments should only be used on separate lines.



---

# 7

## Programming Examples

This chapter consists of five programming examples in assembly language. The first three examples show typical assembly language code sequences; the last two examples show the correspondence between C, a higher-level programming language, and assembly language.

- |           |  |
|-----------|--|
| Example 1 | Calculates the highest bit position set in a passed parameter. A binary search is used to enhance performance.                             |
| Example 2 | Copies bytes from a source location to a destination location. Both locations and the number of bytes to copy are passed in as parameters. |
| Example 3 | Uses Divide Step to divide a 64-bit signed dividend by a 32-bit signed divisor.  |
| Example 4 | Uses a procedure call from a C program to the Assembler to verify that the program is passing the correct argument.                        |
| Example 5 | Shows a C program that generates assembly code to call <code>printf()</code> .   |

---

## 1. Binary Search for Highest Bit Position

The Shift Double and Extract Unsigned instructions are used to implement a binary search. Bits shifted into general register 0 are effectively discarded.

```
.CODE
    .EXPORT post
;
; This procedure calculates the highest bit position
; set in the word passed in as the first argument.
; If passed parameter is non-zero, the algorithm
; starts by assuming it is one.
; A binary search for a set bit is then used
; to enhance performance.
;
; The calculated bit position is returned to the caller.
;
    .PROC
post
    .CALLINFO SAVE_RP
    .ENTER
    COMB,=,N    %r0,%arg0,all_zeros    ; No bits set
    LDI        31,%ret0                ; assume 2 to the 0 power
;
; if extracted bits non-zero, fall thru to change assumption
; else set up 16 low order bits and keep assumption
;
    EXTRU,<>    %arg0,15,16,%r0        ; check 16 high order bits
    SHD,TR     %arg0,%r0,16,%arg0     ; left shift arg0 16 bits
    ADDI      -16,%ret0,%ret0        ; assume 2 to the 16 power
;
; if extracted bits non-zero, fall thru to change assumption
; else set up 8 low order bits and keep assumption
;
    EXTRU,<>    %arg0,7,8,%r0          ; check next 8 high order bits
    SHD,TR     %arg0,%r0,24,%arg0     ; left shift arg0 8 bits
    ADDI      -8,%ret0,%ret0         ; assume 8 higher power of 2
;
; if extracted bits non-zero, fall thru to change assumption
; else set up 4 low order bits and keep assumption
;
    EXTRU,<>    %arg0,3,4,%r0          ; check next 4 high order bits
    SHD,TR     %arg0,%r0,28,%arg0     ; left shift arg0 4 bits
    ADDI      -4,%ret0,%ret0         ; assume 4 higher power of 2
;
; if extracted bits non-zero, fall thru to change assumption
; else set up 2 low order bits and keep assumption
;
    EXTRU,<>    %arg0,1,2,%r0          ; check next 2 high order bits
    SHD,TR     %arg0,%r0,30,%arg0     ; left shift arg0 2 bits
    ADDI      -2,%ret0,%ret0         ; assume 2 higher power of 2
```

**1. Binary Search for Highest Bit Position**

```
;
; if extracted bit is zero, fall thru and keep assumption
; else make conclusion
;
    EXTRU,=    %arg0,0,1,%r0        ; check next bit
    ADDI      -1,%ret0,%ret0        ; next higher power of 2
    B,N       tally
all_zeros
    LDI       -1,%ret0
tally
    .LEAVE
    .PROCEND
```

---

## 2. Copying a String

This example contains a section of assembly code that moves a byte string of arbitrary length to an arbitrary byte address.

```
; The routine reflect copies bytes from the source location
; to the destination location.
;
; The first parameter is the source address and the second
; parameter is the destination address.
;
; The third parameter is the number of bytes to copy.
;
; For performance, larger chunks of bytes are handled differently.
;
        .CODE
        .EXPORT      reflect,ENTRY

reflect
        .PROC
        .CALLINFO    ENTRY_GR=6,SAVE_RP
        .ENTER
        COMB,=,N      %arg2,%r0,fallout      ; done, count is zero
        COMB,<,N      %arg2,%r0,choke       ; caller error, neg count
        OR            %arg0,%arg1,%r6       ; source and dest
        OR            %r6,%arg2,%r6        ; count
        EXTRU,=       %r6,31,2,%r0         ; 2 low order bits = 0?
        B,N           onebyte              ; yes, skip this branch
        ADDIBT,<,N    -16,%arg2,cekchunk    ; no, skip chunkify if count<0

chunkify
        LDWM          16(%arg0),%r6         ; word 1- > temp1
                                                ; point ahead 4 words in source
        LDW           -12(%arg0),%r5       ; place mark 3 wds back- >temp2
        LDW           -8(%arg0),%r4        ; place mark 2 wds back- >temp3
        LDW           -4(%arg0),%r3        ; place mark 1 wds back- >temp4
        STW           %r5,4(%arg1)         ; dest wd 2 <-temp2
        STWM          %r6,16(%arg1)        ; dest wd 1 <-temp1
                                                ; point ahead 4 words in dest
        STW           %r4,-8(%arg1)        ; dest wd 3 <-temp3
        ADDIBF,<      -16,%arg2,chunkify    ; loop if count > 0
        STW           %r3,-4(%arg1)        ; dest wd 2 <-temp1

cekchunk
        ADDIBT,<      12,%arg2,exeunt       ; go if count < -12
        COPY         %r0,%ret0            ; clear rtnval

subchunk
        LDWS,MA       4(%arg0),%r6         ; word- >temp1
                                                ; point ahead 4 bytes in source
        ADDIBF,<      -4,%arg2,subchunk     ; go if count<4
        STWS,MA       %r6,4(%arg1)        ; dest< -temp1
                                                ; point ahead 4 bytes in dest
```

```
        B          exeunt          ; all done
        COPY       %r0,%ret0      ; clear rtnval

onebyte
        LDBS,MA    1(%arg0),%r6    ; templ < -byte,bump src pointer

onemore
        STBS,MA   %r6,1(%arg1)     ; dest<-templ,bump dest pointer
        ADDIBF,=,N -1,%arg2,onemore ; decrement count
        LDBS,MA   1(%arg0),%r6    ; compare for 0.
        LDBS,MA   1(%arg0),%r6    ; delay slot
        LDBS,MA   1(%arg0),%r6    ; templ <-byte,bump src pointer

fallout
        B          exeunt
        COPY       %r0,%ret0

choke
        LDI       14,%ret0

exeunt
        .LEAVE
        .PROCEND
```

---

### 3. Dividing a Double-Word Dividend

This example contains the code sequence to divide a 64-bit signed dividend by a 32-bit signed divisor using the DS (Divide Step) instruction. Table 7-1 on page 135 lists the registers that this program uses.

```

start
    MOVB,>=      dvdu,rem,check_mag ; Move upper dividend
                                   ; check for &< 0
    ADD          0,dvdl,quo         ; Move lower dividend always
    SUB          0,quo,quo          ; Get absolute value of
    SUBB         0,rem,rem          ; the dividend in rem,quo

check_mag
    SUBT,=       0,dvr,tp           ; Check 0, clear carry,
                                   ; negate the divisor
                                   ; and trap if dvr = 0
    DS          0,tp,0;            ; Set V-bit to the complement
                                   ; of the divisor sign
    ADD         quo,quo,quo         ; Shift msb bit into carry
    DS,&<<        rem,dvr,rem        ; 1st divide step, if carry
                                   ; out msb of quotient = 0
    B,n         min_ovfl          ; Abs(quotient) > 2**31
                                   ; deal with elsewhere
    ADDC        quo,quo,quo         ; Shift quo with/into carry
    DS          rem,dvr,rem        ; 2nd divide step
    ; ...
    ;repeat divide step sequence
    ; ...
    ADDC        quo,quo,quo         ; Shift quo with/into carry
    DS          rem,dvr,rem        ; 31st divide step
    ADDC        quo,quo,quo         ; Shift quo with/into carry
    DS          rem,dvr,rem        ; 32nd divide step,
    ADDC        quo,quo,quo         ; Shift last quo bit into quo
    ADDB,>=,n    rem,0,finish       ; Branch if pos. rem
    ADD,&<        dvr,0,0            ; If dvr > 0, add dvr
    ADD,tr      rem,dvr,rem        ; for correcting rem.
    ADDL        rem,tp,rem         ; Else add absolute value dvr

finish
    ADD,>=      dvdu,0,0            ; Set sign of rem
    SUB        0,rem,rem           ; to sign of dividend
    XOR,>=     dvdu,dvr,0          ; Get correct sign of quo
    SUB        0,quo,quo           ; based on operand signs

```

**Table 7-1**      **Register Designations**

<b>Register Designations</b>	<b>Purpose</b>
dvr	Register holding divisor.
dvdu dvdl	Pair of registers holding dividend.
tp	Temporary register.
quo	Register holding quotient.
rem	Register holding remainder.

## 4. Demonstrating the Procedure Calling Convention

A C program calls an assembly language program to test if `.ENTER` and `.LEAVE` are working correctly. The assembly language program checks to see if the C program has passed the value zero in `arg0`. The assembly language program then returns the value `-9` in `ret0` to the calling program.

You need to compile this assembly listing using `cc`. The registers `ret0` and `arg0` are declared within `/usr/lib/pcc_prefix.s`, which is automatically included when you give the C compiler an assembly file.

To remove the dependency on `pcc_prefix.s`, replace all occurrences of `ret0` with `%r28` and `arg0` with `%r26`.

### C Program Listing

```
#include <stdio.h>
int errorcount = 0;
main ()
{
    int toterr = 0;
    printf("TESTING FEATURE 000");
    fflush(stdout);
    if( feat000(000) != -9 ) ++errorcount;
    printf("    %d errors\n",errorcount);
    toterr += errorcount;
    errorcount = 0;
}
```



## Assembly Program Listing

```
; Assembler Module that passes results back to C driver module
myfeat .EQU 000
success .EQU -9

.CODE
.IMPORT errorcount,DATA
.SUBSPA $CODE$
.EXPORT feat000,ENTRY

.PROC
.CALLINFO
feat000 .ENTER
      LDI 0,ret0
      COMIB,<> myfeat,arg0,exit
      NOP
      LDI success,ret0
exit .LEAVE
     .PROCEND
     .END
```

## 5. Output of the cc -S Command

This example shows how a simple C program generates assembly language code. The program calls the `printf()` routine. To run the assembled code, you need to link the file `/usr/ccs/lib/crt0.o` and the C library file. Remember that the `ld` command requires that you link the `crt0.o` file first in 32-bit mode only. You do not need to link `/usr/ccs/lib/crt0.o` in 64-bit mode..

### C Program Listing

```
main ()
{
    printf ("Hello World\n");
}
```

### Assembly Program Listing From the C Compiler

```
.LEVEL 1.0
.SPACE $TEXT$,SORT=8
.SUBSPA $CODE$,QUAD=0,ALIGN=4,ACCESS=44,CODE_ONLY,SORT=24
main
.PROC
.CALLINFO CALLER,FRAME=0,SAVE_RP
.ENTRY
STW    %r2,-20(0,%r30) ;offset 0x0
LDO    48(%r30),%r30    ;offset 0x4
ADDIL  LR'$THIS_DATA$-$global$,%r27    ;offset 0x8
.CALL  ARGW0=GR        ;in=26;
BL     printf,%r2      ;offset 0xc
LDO    RR'$THIS_DATA$-$global$(%r1),%r26 ;offset 0x10
L$exit1
LDW    -68(0,%r30),%r2 ;offset 0x14
BV     %r0(%r2)        ;offset 0x18
.EXIT
LDO    -48(%r30),%r30  ;offset 0x1c
.PROCEND ;

.SPACE $TEXT$
.SUBSPA $LIT$,QUAD=0,ALIGN=8,ACCESS=44,SORT=16
.SUBSPA $CODE$
.SPACE $PRIVATE$,SORT=16
.SUBSPA $DATA$,QUAD=1,ALIGN=8,ACCESS=31,SORT=16
$THIS_DATA$
.ALIGN 4
.STRINGZ "Hello World\n"

.SUBSPA $SHORTDATA$,QUAD=1,ALIGN=8,ACCESS=31,SORT=24
```

```
$THIS_SHORTDATA$  
    .SUBSPA $BSS$,QUAD=1,ALIGN=8,ACCESS=31,ZERO, SORT=82  
$THIS_BSS$  
    .SUBSPA $SHORTBSS$,QUAD=1,ALIGN=8,ACCESS=31,ZERO, SORT=80  
$THIS_SHORTBSS$  
    .SUBSPA $STATICDATA$,QUAD=1,ALIGN=8,ACCESS=31, SORT=16  
$STATIC_DATA$  
    .SUBSPA $SHORTSTATICDATA$,QUAD=1,ALIGN=8,ACCESS=31, SORT=24  
$SHORT_STATIC_DATA$  
    .IMPORT $global$,DATA  
    .SPACE $TEXT$  
    .SUBSPA $CODE$  
    .EXPORT main,ENTRY,PRIV_LEV=3,RTNVAL=GR  
    .IMPORT printf,CODE  
    .END
```

Programming Examples

**5. Output of the cc -S Command**

---

# 8

## Diagnostic Messages

This appendix lists all error messages that originate from the PA-RISC Assembler.

The Assembler error messages are divided into the following categories:

<b>Warning Messages</b>	Conditions that cause errors in program execution.
<b>Error Messages</b>	Conditions that cause the Assembler to terminate abnormally.
<b>Panic Messages</b>	Conditions that cause the Assembler to abort immediately.
<b>User Warnings</b>	Conditions that cause the Assembler to produce an object file and possibly to take a specified corrective action.
<b>Limit Errors</b>	Conditions caused by running into Assembler limits or running out of memory.
<b>Branching Errors</b>	Conditions that prevent the Assembler from creating an object file.

Message descriptions use symbols in the form *&<operand>* to designate Assembler source elements that are the subject of the error. When a message is printed during the assembly operation, the *&<operand>* symbol is replaced by the appropriate source element.

The text for the Assembler messages is stored in a file with the path name:

```
/usr/lib/nls/msg/$LANG/as.cat
```

The default value for the localization environment variable LANG is C.

## Warning Messages

The following messages describe compiler warnings that prevent the Assembler from creating an object file. You must correct these errors to assemble your program.

- |          |                |   |
|----------|----------------|---|
| <b>1</b> | <b>MESSAGE</b> | Use of old style opcode, "%s"   |
|          | <b>CAUSE</b>   | Attempt to use an opcode that has been renamed for PA2.0.               |
|          | <b>ACTION</b>  | Consult the PA-RISC Architecture manual for the new form of the opcode. |
| <b>2</b> | <b>MESSAGE</b> | Unknown option "%s" ignored.  |
|          | <b>CAUSE</b>   | An unknown or invalid command-line option was passed to the assembler.  |
|          | <b>ACTION</b>  | Remove the invalid option from the command-line.                        |
| <b>3</b> | <b>MESSAGE</b> | Argument is missing for "%s"  |
|          | <b>CAUSE</b>   | Missing argument for command-line option <i>&lt;operand&gt;</i> .       |
|          | <b>ACTION</b>  | Add an argument to the command-line option.                             |
| <b>4</b> | <b>MESSAGE</b> | Illegal argument for option "%s"  |
|          | <b>CAUSE</b>   | Invalid value for command-line option <i>&lt;operand&gt;</i>            |
|          | <b>ACTION</b>  | Supply a valid value for the command-line option.                       |
| <b>5</b> | <b>MESSAGE</b> | Usage of field selector "%s" with instruction "%s" may be incorrect     |
|          | <b>CAUSE</b>   | Using an improper field selector for the instruction.                   |
|          | <b>ACTION</b>  | Change the field selector.  |

<b>7</b>	<b>MESSAGE</b>	Space characteristics may not be changed after first declaration
	<b>CAUSE</b>	Attempt to change the values assigned to keywords for a space previously declared with a <code>.SPACE</code> directive. The first declaration of the space may be in the file <code>pcc_prefix.s</code> .
	<b>ACTION</b>	Use desired values for keywords on first declaration of space. Specify keyword values on first declaration of space only.
<b>8</b>	<b>MESSAGE</b>	Subspace characteristics may not be changed after first definition
	<b>CAUSE</b>	Attempt to change the values assigned to keywords for a subspace previously declared with a <code>.SUBSPACE</code> directive. The first declaration of the subspace may be in the file <code>pcc_prefix.s</code> .
	<b>ACTION</b>	Use desired values for keywords on first declaration of subspace. Specify keyword values on first declaration of subspace only.
<b>10</b>	<b>MESSAGE</b>	Alignment omitted - 8 assumed
	<b>CAUSE</b>	Missing argument for <code>.ALIGN</code> directive.
	<b>ACTION</b>	Use a valid power of two integer argument with <code>.ALIGN</code>
<b>11</b>	<b>MESSAGE</b>	Missing value - zero assumed
	<b>CAUSE</b>	Missing argument for <code>.ORIGIN</code> or <code>.EQU</code> directive
	<b>ACTION</b>	Use a valid integer argument.
<b>12</b>	<b>MESSAGE</b>	Size omitted - 4 assumed

Diagnostic Messages  
Warning Messages

	<b>CAUSE</b>	Missing argument for <code>.COMM</code> directive.
	<b>ACTION</b>	Specify the actual number of bytes to reserve.
<b>13</b>	<b>MESSAGE</b>	Modification of <code>%r3</code> with a frame size larger than 8191 bytes violates <code>.ENTER/.LEAVE</code> convention
	<b>CAUSE</b>	General register 3 used as a temporary register within a procedure where the frame size was set by the <code>FRAME</code> keyword to a size larger than 8191 bytes.  Note: <code>%r3</code> is predefined as a frame pointer for procedures with large frames.
	<b>ACTION</b>	Don't use <code>%r3</code> as a temporary register in a procedure that has a large frame.
<b>14</b>	<b>MESSAGE</b>	<code>KEEP</code> should not be in force for this statement
	<b>CAUSE</b>	<code>.KEEP</code> directive used outside of a procedure
	<b>ACTION</b>	Remove the <code>.KEEP</code> directive.
<b>15</b>	<b>MESSAGE</b>	Procedure makes calls but is not flagged as <code>CALLER</code> in <code>CALLINFO</code>
	<b>CAUSE</b>	Missing <code>CALLER</code> keyword in <code>.CALLINFO</code> directive for this procedure.
	<b>ACTION</b>	Add <code>CALLER</code> keyword to <code>.CALLINFO</code> for this procedure.
<b>16</b>	<b>MESSAGE</b>	Redefinition of symbol
	<b>CAUSE</b>	Symbol used in label definition is already defined.



	<b>ACTION</b>	Use a different name in the label part to avoid overwriting the previous definition.
<b>17</b>	<b>MESSAGE</b>	Existing register name, number, and type are being overwritten
	<b>CAUSE</b>	Name (label) used with .REG was previously defined.
	<b>ACTION</b>	Use a different name in the label part to avoid overwriting previous definition.
<b>18</b>	<b>MESSAGE</b>	The "%s" error message catalog cannot be located
	<b>CAUSE</b>	Error message catalog for LANG <operand> cannot be accessed.
	<b>ACTION</b>	Insure that your NLSPATH variable is set correctly. Also insure that the default error message catalog for the assembler is accessible in /usr/lib/nls/msg/C/as.cat
<b>19</b>	<b>MESSAGE</b>	Defining register missing or defining register has no type
	<b>CAUSE</b>	Parameter to .REG is not one of the predefined assembler registers, nor is it a previously defined (by another .REG directive) register.
	<b>ACTION</b>	Either use one of the predefined assembler register names or a register name previously defined by a .REG directive.
<b>20</b>	<b>MESSAGE</b>	General register expected in this field - %s
	<b>CAUSE</b>	Wrong register type used.
	<b>ACTION</b>	Use a general register.
<b>21</b>	<b>MESSAGE</b>	Space register expected in this field - %s

Diagnostic Messages

Warning Messages

	<b>CAUSE</b>	Wrong register type used.
	<b>ACTION</b>	Use a space register.
<b>22</b>	<b>MESSAGE</b>	Control register expected in this field - %s
	<b>CAUSE</b>	Wrong register type used.
	<b>ACTION</b>	Use a control register.
<b>23</b>	<b>MESSAGE</b>	Floating point register expected in this field - %s
	<b>CAUSE</b>	Wrong register type used.
	<b>ACTION</b>	Use a floating-point register.
<b>25</b>	<b>MESSAGE</b>	This subspace should have no initialized data in it
	<b>CAUSE</b>	Use of a directive, such as <code>.WORD</code> that allocates and initializes data, when the currently active subspace has the <code>ZERO</code> keyword associated with it, which disallows initialized data.
	<b>ACTION</b>	Either use the <code>.BLOCK</code> directive to allocate storage in this subspace or remove the <code>ZERO</code> keyword from the <code>.SUBSPACE</code> definition.
<b>26</b>	<b>MESSAGE</b>	Output file name missing
	<b>CAUSE</b>	Filename needed for <code>-o</code> command-line option.
	<b>ACTION</b>	Supply a valid filename for the <code>-o</code> command-line option.
<b>27</b>	<b>MESSAGE</b>	XREF file name missing after <code>-v</code>
	<b>CAUSE</b>	Filename needed for <code>-v</code> command-line option.
	<b>ACTION</b>	Supply a valid filename for the <code>-v</code> command-line option.
<b>28</b>	<b>MESSAGE</b>	Only one copyright message permitted

	<b>CAUSE</b>	More than one <code>.COPYRIGHT</code> directive encountered.
	<b>ACTION</b>	Remove the extra <code>.COPYRIGHT</code> directive(s).
<b>29</b>	<b>MESSAGE</b>	A procedure may not be empty
	<b>CAUSE</b>	Procedure with no code encountered.
	<b>ACTION</b>	Add at least one instruction to procedure or delete the procedure from the source file.
<b>30</b>	<b>MESSAGE</b>	Procedure does not have <code>.CALLINFO</code>
	<b>CAUSE</b>	Procedure requires a <code>.CALLINFO</code> due to use of <code>.ENTER/</code> <code>.LEAVE</code> or if there are no unwind space requirements.
	<b>ACTION</b>	Insert a correct <code>.CALLINFO</code> directive at the start of the procedure.
<b>31</b>	<b>MESSAGE</b>	Empty source file(s)
	<b>CAUSE</b>	Input source file was empty.
	<b>ACTION</b>	An empty source file usually indicates an error of some kind.
<b>32</b>	<b>MESSAGE</b>	Missing <code>.PROCEND</code>
	<b>CAUSE</b>	A <code>.PROC</code> was not terminated by a corresponding <code>.PROCEND</code>
	<b>ACTION</b>	Insert a <code>.PROCEND</code>
<b>33</b>	<b>MESSAGE</b>	Cache hint may not work on some hardware
	<b>CAUSE</b>	An instruction was encountered that was performing a load to <code>%r0</code> . This type of instruction is reserved as a hint to the cache system. However some of the early hardware didn't properly perform this cache hint feature.

Diagnostic Messages

Warning Messages

	<b>ACTION</b>	Avoid using loads to %r0 if you want your code to execute correctly on all PA-RISC machines.
<b>34</b>	<b>MESSAGE</b>	Missing .LEVEL directive; A .LEVEL 1.0 was inserted before .ALLOW
	<b>CAUSE</b>	A .ALLOW directive was encountered without first seeing a .LEVEL directive.
	<b>ACTION</b>	Insert a .LEVEL directive at the start of the source file.
<b>35</b>	<b>MESSAGE</b>	Use of .ALLOW %s not meaningful for file assembled at .LEVEL %s
	<b>CAUSE</b>	The .ALLOW directive supplied has no meaning for a file assembled at given .LEVEL. For example if your source file specifies .LEVEL 1.1 then a .ALLOW 1.0 or .ALLOW 1.1 would not be meaningful for this source file.
	<b>ACTION</b>	Removed .ALLOW directive or change .LEVEL.
<b>36</b>	<b>MESSAGE</b>	Use of %s is incorrect for the current %s of %s
	<b>CAUSE</b>	Use of a feature that is not supported on the hardware that the assembler is targeting. The assembler is told what hardware it is targeting through the use of the .LEVEL and .ALLOW directives. Using PA1.1 features while at .LEVEL 1.0 will generate this message.
	<b>ACTION</b>	Insert a .LEVEL or .ALLOW directive as appropriate, or remove the offending instruction from the source file.

37	<b>MESSAGE</b>	Use of %s requires key bit %s to be enabled with a .LEVEL or .ALLOW
	<b>CAUSE</b>	Use of a feature that requires a key bit to be set.
	<b>ACTION</b>	Add key bit to the .LEVEL or .ALLOW directive.
38	<b>MESSAGE</b>	Encoding for %s requires a format that is incorrect for the current %s of %s
	<b>CAUSE</b>	The instruction requires an encoding format that is not available on the hardware that the assembler is targeting. The assembler is told what hardware it is targeting through the use of the .LEVEL and .ALLOW directives. Using a PA-RISC 2.0 instruction encoding format while at .LEVEL 1.1 will generate this message.
	<b>ACTION</b>	Change the instruction such that it can be properly encoded on the hardware that the assembler is targeting or change the .LEVEL to 2.0.
39	<b>MESSAGE</b>	The completer specified ,%s is obsolete but will be accepted as ,%s
	<b>CAUSE</b>	The use of the ,SH completer is incorrect.
	<b>ACTION</b>	Replace the ,SH completer by the ,BC completer is the source.
40	<b>MESSAGE</b>	Poorly formed operand, accepted as %s(%r0)
	<b>CAUSE</b>	The previous assembler would silently accept poorly formed operands and add a trailing (%r0) to

		the operand. The new assembler will also accept these poorly formed operands and also add the trailing (%r0) to the operand, but it additionally flags this as a potential problem by issuing this warning.
	<b>ACTION</b>	Add a trailing (%r0) to the operand or otherwise correct the poorly formed operand.
<b>41</b>	<b>MESSAGE</b>	A register typed operand is expected here - %s
	<b>CAUSE</b>	The old assembler would accept integers between 0 and 31 as valid registers, or alternatively accept names that were defined using a .EQU to an integer. The new assembler performs additional type checks on the operands and wants to see either a predefined register (starting with a %) or a register name that was previously defined by the .REG directive. The file /usr/lib/pcc_prefix.s is typically included when the assembler is invoked through cc(1) and this file defines many register names that are typically used in assembly programs.
	<b>ACTION</b>	Inspect the source file and replace the integer with the appropriate predefined register, or for a name, find and replace the .EQU directive with a correct .REG directive.  Alternatively the w41 command-line option may be used to suppress all occurrences of this warning message.  Note that future versions of the assembler may require proper register typing of operands in order

		for it to be able to disambiguate between immediates and registers for certain opcodes. Thus in future versions of the assembler this message may be a non-suppressible error instead of a warning.
<b>42</b>	<b>MESSAGE</b>	Use of %s is incorrect for the current %s of %s
	<b>CAUSE</b>	Use of a cbit in a FTEST or FCMP instruction, with a .LEVEL of 1.1 or 1.0. The cbit feature of FCMP and FTEST is only available for .LEVEL 2.0.
	<b>ACTION</b>	Insert a .LEVEL or .ALLOW directive as appropriate, or remove the offending instruction from the source file.
<b>44</b>	<b>MESSAGE</b>	Value for ACCESS was not specified for this new .SUBSPA directive.
	<b>CAUSE</b>	A new subspace is being defined by the this .SUBSPA directive and a value of the ACCESS keyword is not supplied. You must always give a value for ACCESS when defining a new subspace.
	<b>ACTION</b>	If the subspace that you are referencing is a predefined subspace then make sure that the header file /usr/lib/pcc_prefix.s is being included. You must add the header file to the command-line if you are invoking as(1) directly. The header file will typically be included for you if you invoke the assembler using cc(1)  If you are declaring a new subspace then add the proper ACCESS definition to the .SUBSPA directive.

Diagnostic Messages

Warning Messages

		The value <code>ACCESS=0x2c</code> should be used for all code or read-only subspaces and the value <code>ACCESS=0x1f</code> should be used for all data or read-write subspaces.
<b>45</b>	<b>MESSAGE</b>	Previous value for <code>.SHLIB</code> is being changed to this value
	<b>CAUSE</b>	More than one <code>.SHLIB</code> directive was encountered.
	<b>ACTION</b>	Remove the extra <code>.SHLIB</code> directive(s).
<b>46</b>	<b>MESSAGE</b>	The <code>+DA</code> option conflicts with the <code>.LEVEL</code> directive, using <code>.LEVEL %s</code> .
	<b>CAUSE</b>	The assembly source file contained a <code>.LEVEL</code> directive which conflicted with the command-line <code>+DA</code> option. The assembler always honors the <code>.LEVEL</code> directive found in the source file. The command-line <code>+DA</code> was ignored.
	<b>ACTION</b>	Remove <code>+DA</code> from the command-line used to invoke the assembler.
<b>47</b>	<b>MESSAGE</b>	The behavior of instruction <code>%s</code> is undefined with the operands and completers supplied.
	<b>CAUSE</b>	The behavior of the instruction used is undefined with the operands supplied.
	<b>ACTION</b>	Read the PA-RISC Architecture manual entry for the instruction being used, paying attention to the cases in which the behavior of the instruction is undefined. Recode the operands for the instruction so that the operands don't cause the instruction to be undefined.



<b>48</b>	<b>MESSAGE</b>	Expression encountered while expecting a register; Register %s substituted for expression.
	<b>CAUSE</b>	An expression was encountered in a location where a predefined register (starting with a % or a register name that was previously defined by the .REG directive was expected. (Also see warning 41).
	<b>ACTION</b>	Replace expression with a predefined register or a register name that was previously defined by the .REG directive.
<b>49</b>	<b>MESSAGE</b>	Number is too large. The value -1 will be used instead.
	<b>CAUSE</b>	A number was encountered that was too large to fit in a 64-bit integer.
	<b>ACTION</b>	Replace the number with a smaller value.

## Error Messages

The following messages describe compilation errors that prevent the Assembler from creating an object file. You must correct these errors to assemble your program.

<b>1000</b>	<b>MESSAGE</b>	Unterminated quoted string
	<b>CAUSE</b>	String specified was missing the trailing double quote. Strings literals can not span multiple lines.
	<b>ACTION</b>	Add trailing double quote to string.
<b>1001</b>	<b>MESSAGE</b>	Undefined register symbol
	<b>CAUSE</b>	Name specified is not a predefined or .REG defined register symbol
	<b>ACTION</b>	Correct spelling of name or add .REG directive to define name.
<b>1002</b>	<b>MESSAGE</b>	Undefined completer
	<b>CAUSE</b>	Invalid value used in completer field.
	<b>ACTION</b>	Use a proper completer as specified in <i>PA-RISC Architecture and Instruction Set Reference Manual</i> .
<b>1003</b>	<b>MESSAGE</b>	Improper completer ,%s specified for opcode %s
	<b>CAUSE</b>	The completer used is not a valid completer for the instruction.
	<b>ACTION</b>	Consult the <i>PA-RISC Architecture and Instruction Set Reference Manual</i> for the list of valid completers for this instruction.
<b>1004</b>	<b>MESSAGE</b>	Illegal completer combination specified for opcode %s
	<b>CAUSE</b>	The combination of completers used is not valid for the instruction.

	<b>ACTION</b>	Consult the <i>PA-RISC Architecture and Instruction Set Reference Manual</i> , for the list of valid completer combinations for this instruction.
<b>1005</b>	<b>MESSAGE</b>	Unable to open xref file: %s
	<b>CAUSE</b>	Assembler could not create or access the file specified with the <code>-v</code> command-line option.
	<b>ACTION</b>	Insure that the directory is writable.
<b>1007</b>	<b>MESSAGE</b>	Label not allowed here in this expr
	<b>CAUSE</b>	Assembler will not allow a label here.
	<b>ACTION</b>	Expressions must be in the form <i>label1[[-label2]+constant_exp]</i> Reorder expression to be in the proper form.
<b>1008</b>	<b>MESSAGE</b>	Illegal symbol in expression
	<b>CAUSE</b>	An expression contains a sequence other than <i>label operator term</i> or <i>term operator term</i>
	<b>ACTION</b>	Place operators <code>+</code> , <code>-</code> , between a label and a term, or place operators <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> between a term and term. Note that a term means a constant expression.
<b>1009</b>	<b>MESSAGE</b>	Field selector not allowed in pc-relative expression
	<b>CAUSE</b>	Field selector, such as <code>L'</code> or <code>R'</code> used on a expression that is a branch target.
	<b>ACTION</b>	Omit field selectors from branch target expression.
<b>1010</b>	<b>MESSAGE</b>	String not allowed in pc_relative expression
	<b>CAUSE</b>	String used for branch target expression.

Diagnostic Messages  
Error Messages

	<b>ACTION</b>	Replace with an expression beginning with a label or “.”
<b>1011</b>	<b>MESSAGE</b>	“.” allowed in <code>pc_rel</code> expression only
	<b>CAUSE</b>	A period (.) used as an operand in a non-branch context, or used as a target in external branch or vectored branch.
	<b>ACTION</b>	Use “.” only for pc-relative branches, not in branch external or branch vectored.
<b>1012</b>	<b>MESSAGE</b>	PC-relative expression must begin with . or label
	<b>CAUSE</b>	Branch target is poorly formed.
	<b>ACTION</b>	Use a label or “.” as the first term of a branch expression
<b>1013</b>	<b>MESSAGE</b>	Second label not allowed in <code>pc_relative</code> expression
	<b>CAUSE</b>	Branch target is poorly formed: <i>label operator label</i>
	<b>ACTION</b>	Use an offset in place of the second label.
<b>1014</b>	<b>MESSAGE</b>	Labels may not be added, they may only be subtracted
	<b>CAUSE</b>	Attempt to form the sum of two labels.
	<b>ACTION</b>	Use an offset in place of the second label.
<b>1015</b>	<b>MESSAGE</b>	Unexpected end of expression
	<b>CAUSE</b>	Nothing follows a +, -, /, or * in an expression.
	<b>ACTION</b>	Place meaningful terms, integers or labels after operator.

<b>1016</b>	<b>MESSAGE</b>	General register %s is out of range
	<b>CAUSE</b>	Register number specified greater than 31 or less than 0.
	<b>ACTION</b>	Use a valid general register number between 0 and 31.
<b>1017</b>	<b>MESSAGE</b>	Value of %s for space register not in %sr0..%sr3
	<b>CAUSE</b>	Space register specified was not in the legal range.
	<b>ACTION</b>	Use a space register in the valid range 0 to 3.
<b>1018</b>	<b>MESSAGE</b>	Value of %s for space register not in %sr0..%sr7
	<b>CAUSE</b>	Space register number specified greater than 7 or less than 0
	<b>ACTION</b>	Use a valid space register number between 0 and 7.
<b>1019</b>	<b>MESSAGE</b>	Opcode %s not defined
	<b>CAUSE</b>	Characters in the opcode field do not comprise a legal machine instruction or directive.
	<b>ACTION</b>	Starting in column 2, use only valid opcodes and directives as specified in <i>PA-RISC Architecture and Instruction Set Reference Manual</i> .
<b>1020</b>	<b>MESSAGE</b>	Number required for keyword value
	<b>CAUSE</b>	A .CALLINFO keyword was assigned a non-numeric argument.
	<b>ACTION</b>	Ensure that .CALLINFO keywords are assigned numeric or register values.
<b>1021</b>	<b>MESSAGE</b>	Unrecognized value for keyword

Diagnostic Messages  
Error Messages

	<b>CAUSE</b>	Illegal assignment for ARGW or RTNVAL keyword in .CALL or .EXPORT directive
	<b>ACTION</b>	Use NO, GR, FR, or FU as appropriate.
<b>1022</b>	<b>MESSAGE</b>	This directive must occur within a declared subspace: %s
	<b>CAUSE</b>	The directive used must appear inside of a .SUBSPA. The directive was place outside of a subspace.
	<b>ACTION</b>	Insert a .SUBSPA before issuing this directive.
<b>1023</b>	<b>MESSAGE</b>	Directive %s not allowed inside a procedure
	<b>CAUSE</b>	Use of this directive must occur outside of a .PROC The .LOCCT, .SPACE or .SUBSPA directives must occur outside of a .PROC
	<b>ACTION</b>	Do not attempt to change the location counter, space or subspace within a procedure.
<b>1024</b>	<b>MESSAGE</b>	Space name required
	<b>CAUSE</b>	.SPACE directive is not followed by a valid name.
	<b>ACTION</b>	Follow .SPACE directive with a valid name.
<b>1025</b>	<b>MESSAGE</b>	Unrecognized keyword
	<b>CAUSE</b>	Directive, such as .SPACE, .SUBSPA, or .CALLINFO followed by a keyword not specified in the Assembler manual.
	<b>ACTION</b>	Follow directives with legal keywords separated by commas.
<b>1026</b>	<b>MESSAGE</b>	Identifier %s previously defined

	<b>CAUSE</b>	The name being defined already has a definition. Each identifier can only have one legal meaning. You can't define both a space and a subspace with the same name.
	<b>ACTION</b>	Change the name used by one of the definitions to a unique name.
<b>1027</b>	<b>MESSAGE</b>	This item must be declared within a space
	<b>CAUSE</b>	A directive such as .SUBSPA, is used before the first .SPACE directive.
	<b>ACTION</b>	Insert a valid .SPACE directive prior to the offending directive.
<b>1028</b>	<b>MESSAGE</b>	Subspace name required
	<b>CAUSE</b>	.SUBSPA directive is not followed by a valid name.
	<b>ACTION</b>	Follow .SUBSP directive with a valid name.
<b>1029</b>	<b>MESSAGE</b>	Directive .%s must occur within a procedure
	<b>CAUSE</b>	Directive such as .CALLINFO, .ENTER or .LEAVE is used outside of a procedure. A procedure is defined by a matching pair of .PROC, .PROCEND directives.
	<b>ACTION</b>	Move the offending directive inside a procedure or correct the location of the .PROC and .PROCEND directives for this procedure.
<b>1030</b>	<b>MESSAGE</b>	Only one .CALLINFO per procedure
	<b>CAUSE</b>	Multiple .CALLINFO directives were found within a procedure. A procedure is defined by a matching pair of .PROC, .PROCEND directives.

Diagnostic Messages  
Error Messages

	<b>ACTION</b>	Remove the duplicate <code>.CALLINFO</code> directive or correct the location of the <code>.PROC</code> and <code>.PROCEND</code> directives for this procedure.
<b>1031</b>	<b>MESSAGE</b>	Value for %s must be >=0
	<b>CAUSE</b>	1. In a <code>.CALLINFO</code> directive the parameter <code>FRAME</code> is assigned a negative value. 2. In <code>.BLOCK</code> or <code>.BLOCKZ</code> directive the parameter is a negative value.
	<b>ACTION</b>	Supply a non-negative value.
<b>1032</b>	<b>MESSAGE</b>	Value for %s must be in range %r3..%r18
	<b>CAUSE</b>	In a <code>.CALLINFO</code> directive the parameter <code>ENTRY_GR</code> is assigned an invalid general register.
	<b>ACTION</b>	Use a general register in the range %r3 to %r18 when assigning to the parameter <code>ENTRY_GR</code> .
<b>1033</b>	<b>MESSAGE</b>	Value for %s must be in range %fr12..%fr21
	<b>CAUSE</b>	In a <code>.CALLINFO</code> directive the parameter <code>ENTRY_FR</code> is assigned an invalid floating-point register.
	<b>ACTION</b>	Use a floating-point register in the range %fr12 to %fr21 when assigning to the parameter <code>ENTRY_FR</code> .
<b>1034</b>	<b>MESSAGE</b>	<code>ENTRY_SR</code> must be %sr3 or not specified
	<b>CAUSE</b>	In a <code>.CALLINFO</code> directive the parameter <code>ENTRY_SR</code> specifies an invalid space register.



	<b>ACTION</b>	Only %sr3 is saved using the PA-RISC calling convention. Either specify %sr3 or omit the ENTRY_SR keyword.
<b>1035</b>	<b>MESSAGE</b>	Instructions must occur within a declared subspace: %s
	<b>CAUSE</b>	Instructions present before .SUBSPA directive
	<b>ACTION</b>	Use .SUBSPA directive before issuing instructions
<b>1036</b>	<b>MESSAGE</b>	Illegal use of %previous_sp, must be used as a base register
	<b>CAUSE</b>	The pseudo register %previous_sp is being used in an instruction as a source or destination register. This special register can only be used as a base register.
	<b>ACTION</b>	Use %previous_sp as the base register in a Load or Store instruction.
<b>1037</b>	<b>MESSAGE</b>	nested .PROC
	<b>CAUSE</b>	A second .PROC directive was encountered before a .PROCEND directive
	<b>ACTION</b>	Insert .PROCEND directive before the second .PROC directive, or remove unnecessary .PROC directive.
<b>1038</b>	<b>MESSAGE</b>	Label name required for %s
	<b>CAUSE</b>	Directive, such as .COMM, .REG, .EQU or .MACRO requires that a label be present.
	<b>ACTION</b>	Add a label starting in column 1 to use as the name being defined by this directive.

Diagnostic Messages  
Error Messages

<b>1039</b>	<b>MESSAGE</b>	Missing string constant
	<b>CAUSE</b>	Directive, such as <code>.STRING</code> , <code>.STRINGZ</code> , <code>.VERSION</code> , or <code>.COPYRIGHT</code> is present without a string operand.
	<b>ACTION</b>	Add missing quoted string after directive.
<b>1041</b>	<b>MESSAGE</b>	Name required for %s
	<b>CAUSE</b>	Directive, such as <code>.IMPORT</code> , <code>.EXPORT</code> , <code>.SPACE</code> or <code>.SUBSPA</code> is not followed by an valid identifier.
	<b>ACTION</b>	Follow directive with a legal identifier.
<b>1044</b>	<b>MESSAGE</b>	Name required for label definition
	<b>CAUSE</b>	<code>.LABEL</code> directive is not followed by a legal identifier
	<b>ACTION</b>	Add missing identifier after <code>.LABEL</code> directive.
<b>1045</b>	<b>MESSAGE</b>	Name to be defined by <code>.LABEL</code> must appear as operand.
	<b>CAUSE</b>	The <code>.LABEL</code> directive cannot have an identifier in column one.
	<b>ACTION</b>	Place identifier after <code>.LABEL</code> directive, not before it.
<b>1046</b>	<b>MESSAGE</b>	Duplicate definition of symbol
	<b>CAUSE</b>	The same identifier appeared more than once in column one, or was defined more than one time with a <code>.LABEL</code> directive.
	<b>ACTION</b>	Rename one of the labels.
<b>1047</b>	<b>MESSAGE</b>	Unmatched <code>.PROCEND</code>

	<b>CAUSE</b>	Two .PROCEND directives were encountered without a .PROC directive in between.
	<b>ACTION</b>	Each procedure should begin with a single .PROC and end with a single .PROCEND.
<b>1048</b>	<b>MESSAGE</b>	Comma expected
	<b>CAUSE</b>	A directive which expects two or more operands was missing a comma between it's operands.
	<b>ACTION</b>	Insert comma between operands.
<b>1050</b>	<b>MESSAGE</b>	Illegal symbol in label position
	<b>CAUSE</b>	Illegal character present in identifier which begins in column one.
	<b>ACTION</b>	Use only legal identifiers in label field.
<b>1051</b>	<b>MESSAGE</b>	Illegal symbol in opcode position
	<b>CAUSE</b>	A sequence of characters starting in column two or beyond does not begin with a alphabetic character or period.
	<b>ACTION</b>	Use only valid opcodes and directives starting in column two or beyond.
<b>1052</b>	<b>MESSAGE</b>	Directive name not recognized
	<b>CAUSE</b>	A sequence of characters starting in column two or beyond, beginning with a period, does not correspond to a legal directive.
	<b>ACTION</b>	Check spelling of directive. Use only legal directive names starting in column two or beyond.
<b>1053</b>	<b>MESSAGE</b>	Displacement must be a constant expression

Diagnostic Messages  
Error Messages

	<b>CAUSE</b>	The displacement for this instruction must be a constant expression.
	<b>ACTION</b>	Rewrite the instruction so that it uses a constant expression.
<b>1054</b>	<b>MESSAGE</b>	Unexpected items at end of line
	<b>CAUSE</b>	Legal operands are followed by trailing characters or operators.
	<b>ACTION</b>	Examine entire sequence of operations for syntactic integrity. Possibly insert a “;” to denote a comment after legal operands.
<b>1055</b>	<b>MESSAGE</b>	Label must be defined within a declared subspace
	<b>CAUSE</b>	Label is present before a .SUBSPA directive.
	<b>ACTION</b>	Place label after issuing .SUBSPA directive.
<b>1056</b>	<b>MESSAGE</b>	Poorly formed .DWORD argument
	<b>CAUSE</b>	The syntax for this .DWORD argument is invalid.
	<b>ACTION</b>	Consult the Assembler manual for the valid syntax.
<b>1057</b>	<b>MESSAGE</b>	Unexpected register symbol %s found in a constant expression
	<b>CAUSE</b>	A predefined register or a name defined by a .REG directive was encountered in a location where only an integer constant, a name defined by a .EQU directive, may occur.
	<b>ACTION</b>	Replace the predefined register or a name defined by a .REG directive by a constant, or change the .REG directive into a .EQU directive.

<b>1059</b>	<b>MESSAGE</b>	Divide by zero
	<b>CAUSE</b>	Attempt to perform division with a zero divisor.
	<b>ACTION</b>	Examine definition of divisor, ensure that it is not zero.
<b>1060</b>	<b>MESSAGE</b>	Argument 0 or 2 in FARG upper
	<b>CAUSE</b>	Using the FU value with ARGW0 or ARGW2 keywords.
	<b>ACTION</b>	Only use the FU value with ARGW0 or ARGW2 keywords.
<b>1061</b>	<b>MESSAGE</b>	Closing parenthesis is missing in expression
	<b>CAUSE</b>	Mismatched parenthesis.
	<b>ACTION</b>	Insert closing parenthesis in the expression.
<b>1062</b>	<b>MESSAGE</b>	Macro parameters must be separated by commas
	<b>CAUSE</b>	Formal parameters to .MACRO or actual parameters to a macro invocation are not separated by commas.
	<b>ACTION</b>	Insert commas between parameters.
<b>1063</b>	<b>MESSAGE</b>	Unterminated macro definition
	<b>CAUSE</b>	A .MACRO directive is not matched with a .ENDM directive.
	<b>ACTION</b>	Terminate macro definition with .ENDM.
<b>1064</b>	<b>MESSAGE</b>	Poorly formed macro parameter
	<b>CAUSE</b>	Formal parameter to the macro definition is not in the format accepted by the Assembler.
	<b>ACTION</b>	Change the form of the formal parameter to an acceptable form.

Diagnostic Messages  
Error Messages

<b>1065</b>	<b>MESSAGE</b>	Poorly formed .FLOAT or .DOUBLE argument
	<b>CAUSE</b>	The floating-point number that was used as the argument to .FLOAT or .DOUBLE is not in the right format.
	<b>ACTION</b>	Use a properly formatted floating-point constant for the argument.
<b>1066</b>	<b>MESSAGE</b>	Poorly formed bit field specifier
	<b>CAUSE</b>	Bit field is not specified in the form {x..y} where x and y are non-negative integers.
	<b>ACTION</b>	Specify bit fields in the correct format.
<b>1067</b>	<b>MESSAGE</b>	Bit field too wide for instruction field
	<b>CAUSE</b>	Mismatched bit field declaration and usage.
	<b>ACTION</b>	Use the same length bit field for the bit field being assigned to and the bit field being assigned from.
<b>1068</b>	<b>MESSAGE</b>	Brace outside of macro definition
	<b>CAUSE</b>	Opening brace ({} or closing brace (}) used outside a macro definition. These symbols can only be used with a macro definition. They are used to form a bit field within a macro definition.
	<b>ACTION</b>	Remove opening or closing brace and ensure that they are only used within a macro definition.
<b>1069</b>	<b>MESSAGE</b>	Equal sign required in bit field assignment

	<b>CAUSE</b>	Missing assignment operator = for assigning one bit field to another.
	<b>ACTION</b>	Insert assignment operator = for bit field assignment.
<b>1070</b>	<b>MESSAGE</b>	Bit range must be within {0..31}
	<b>CAUSE</b>	Range specified in bit field is not in the legal range.
	<b>ACTION</b>	Ensure bit field range is within the range 0 to 31.
<b>1071</b>	<b>MESSAGE</b>	Opening brace expected in bit range designator
	<b>CAUSE</b>	Missing opening brace to specify bit field
	<b>ACTION</b>	Use correct format for bit field specification.
<b>1072</b>	<b>MESSAGE</b>	Ending brace expected in bit range designator
	<b>CAUSE</b>	Missing closing brace to specify bit field
	<b>ACTION</b>	Use correct format for bit field specification.
<b>1073</b>	<b>MESSAGE</b>	Unmatched .ENDM
	<b>CAUSE</b>	No .MACRO directive was recognized as corresponding to the .ENDM directive.
	<b>ACTION</b>	Either remove the unmatched .ENDM directive or insert a .MACRO directive in an appropriate position preceding the .ENDM directive.
<b>1074</b>	<b>MESSAGE</b>	Illegal expression type for plabel
	<b>CAUSE</b>	More than one label was found in a plabel expression.

Diagnostic Messages  
Error Messages

	<b>ACTION</b>	Use only one label in a label expression.
<b>1075</b>	<b>MESSAGE</b>	Undefined field selector
	<b>CAUSE</b>	Illegal field selector is being used.
	<b>ACTION</b>	Use correct field selector.
<b>1076</b>	<b>MESSAGE</b>	Recursive macro expansion for %s
	<b>CAUSE</b>	Recursive macro expansion not permitted.
	<b>ACTION</b>	Change macro definition to not be recursive.
<b>1077</b>	<b>MESSAGE</b>	A .CALLINFO may specify either CALLS or NO_CALLS but not both
	<b>CAUSE</b>	A .CALLINFO directive contained both the CALLS and NO_CALLS operands.
	<b>ACTION</b>	Remove either CALLS or NO_CALLS from the .CALLINFO directive.
<b>1078</b>	<b>MESSAGE</b>	Only one architecture revision level can be specified with .LEVEL
	<b>CAUSE</b>	More than one .LEVEL directive was encountered and they specified different architecture levels.
	<b>ACTION</b>	Remove the inappropriate .LEVEL directive.
<b>1079</b>	<b>MESSAGE</b>	Missing "=" for parameter %s
	<b>CAUSE</b>	An operand to a directive is expecting a value to be supplied. For example, the FRAME keyword to the .CALLINFO directive expects an argument to be associated like this: FRAME=32



	<b>ACTION</b>	Supply the missing value to be associated with the keyword using the format <i>keyword=value</i> .
<b>1080</b>	<b>MESSAGE</b>	Missing integer for parameter %s
	<b>CAUSE</b>	An operand to a directive is expecting a value to be supplied. For example, the <code>FRAME</code> keyword to the <code>.CALLINFO</code> directive expects an argument to be associated like this: <code>FRAME=32</code>
	<b>ACTION</b>	Supply the missing value to be associated with the keyword using the format <code>keyword=integer</code> .
<b>1081</b>	<b>MESSAGE</b>	Missing register for parameter %s
	<b>CAUSE</b>	An operand to a directive is expecting a value to be supplied. For example the <code>ENTRY_GR</code> keyword to the <code>.CALLINFO</code> directive expects an argument to be associated like this: <code>ENTRY_GR=%r6</code>
	<b>ACTION</b>	Supply the missing value to be associated with the keyword using the format <code>keyword=register</code> .
<b>1082</b>	<b>MESSAGE</b>	Wrong kind of register for parameter %s
	<b>CAUSE</b>	An operand to a directive is expecting a different register class to be supplied. For example the <code>ENTRY_GR</code> keyword to the <code>.CALLINFO</code> directive expects a general purpose register to be associated; supplying a floating-point register to <code>ENTRY_GR</code> would produce this error message.
	<b>ACTION</b>	Use a register of the expected class for the keyword.

Diagnostic Messages  
Error Messages

<b>1083</b>	<b>MESSAGE</b>	Floating point register %s is out of range
	<b>CAUSE</b>	The value supplied does not correspond to a legal floating-point register number.
	<b>ACTION</b>	Supply a valid floating-point register number.
<b>1084</b>	<b>MESSAGE</b>	Control register %s is out of range
	<b>CAUSE</b>	The value supplied does not correspond to a legal control register number.
	<b>ACTION</b>	Supply a valid control register number.
<b>1085</b>	<b>MESSAGE</b>	Illegal value for alignment
	<b>CAUSE</b>	The value supplied for alignment is not a valid alignment value. Valid values are powers of 2 that are greater than zero and less than or equal to 4096.
	<b>ACTION</b>	Change alignment value to a legal value.
<b>1086</b>	<b>MESSAGE</b>	Only one type can be specified for a symbol with .EXPORT
	<b>CAUSE</b>	More than one symbol type was given for the .EXPORT directive.
	<b>ACTION</b>	Remove the extra type from the .EXPORT directive.
<b>1087</b>	<b>MESSAGE</b>	Bad value for .SHLIB parameter, format is "mm/yyyy"
	<b>CAUSE</b>	The parameter to the .SHLIB directive was not in the proper format. The format must be <i>mm/yyyy</i> , when <i>mm</i> is the integer

		value for the month, (Jan=1,...Dec=12) and <i>yyyy</i> is the year.
	<b>ACTION</b>	Change the parameter to <code>.SHLIB</code> to a valid value.
<b>1088</b>	<b>MESSAGE</b>	Floating point register <i>%s</i> is out of range for <i>%s</i> ,SGL
	<b>CAUSE</b>	The floating-point register used is not valid for this opcode. The multiops FMPYADD,SGL and FMPYSUB,SGL require that the operands be single precision floating-point registers in the range <i>%r16L,%r16R ..</i> <i>%r31L,%r31R</i> . Specifying a register below <i>%r16</i> will result in this error message.
	<b>ACTION</b>	You must use single precision floating-point registers in the range <i>%r16L,%r16R ..</i> <i>%r31L,%r31R</i> when using single precision FMPYADD or FMPYSUB.
<b>1089</b>	<b>MESSAGE</b>	Revision level must be specified for <code>.LEVEL</code> directive
	<b>CAUSE</b>	A <code>.LEVEL</code> directive was missing an architecture level value.
	<b>ACTION</b>	You must specify a valid architecture level of 1.0, 1.1, or 2.0.
<b>1090</b>	<b>MESSAGE</b>	Value of <i>%1s</i> must be in the range [ <i>%2s</i> .. <i>%3s</i> ]
	<b>CAUSE</b>	The immediate value <i>%1s</i> is constrained to be in the range <i>%2s</i> .. <i>%3s</i> for the current instruction.
	<b>ACTION</b>	Use a different instruction that allows for a larger range, or use a register to hold the value <i>%1s</i> .

Diagnostic Messages  
Error Messages

<b>1091</b>	<b>MESSAGE</b>	Incorrect register %s used with %s optional target register must be %s.
	<b>CAUSE</b>	This opcode takes an optional register operand as specified. The register you used for this operand was incorrect.
	<b>ACTION</b>	Use either the correct operand or no operand.
<b>1094</b>	<b>MESSAGE</b>	Value for %s must be in range 0..3
	<b>CAUSE</b>	The value supplied for the privilege level was outside the legal range of 0 to 3.
	<b>ACTION</b>	Supply a valid value in the range 0 to 3 for the privilege level.
<b>1095</b>	<b>MESSAGE</b>	Not enough operands for instruction %s
	<b>CAUSE</b>	While parsing the operands for the current instruction <inst>, the assembler encountered the end of the line while it was still expecting one or more operands for the instruction.
	<b>ACTION</b>	Supply the missing operand(s).
<b>1096</b>	<b>MESSAGE</b>	Missing completer for opcode %s
	<b>CAUSE</b>	The instruction <inst> requires a completer. For example the instruction BB always requires a completer (either < or >=); omitting this completer will cause the assembler to generate this error message.
	<b>ACTION</b>	Inspect the current instruction for missing completers and add the appropriate completer.

<b>1097</b>	<b>MESSAGE</b>	A "(" was expected while parsing the operands of instruction %s
	<b>CAUSE</b>	The operands for the current instruction do not have a "(" in the correct location.
	<b>ACTION</b>	Inspect the current instruction and change the operands to the instruction so that the are valid.
<b>1098</b>	<b>MESSAGE</b>	A ")" was expected while parsing the operands of instruction %s
	<b>CAUSE</b>	The operands for the current instruction do not have a ")" in the correct location.
	<b>ACTION</b>	Inspect the current instruction and change the operands to the instruction so that the are valid.
<b>1099</b>	<b>MESSAGE</b>	A register was expected while parsing the operands of instruction %s
	<b>CAUSE</b>	The operands for the current instruction do not have a register in the correct location.
	<b>ACTION</b>	Inspect the current instruction and change the operands to the instruction so that the are valid.
<b>1100</b>	<b>MESSAGE</b>	A "," was expected while parsing the operands of instruction %s
	<b>CAUSE</b>	The operands for the current instruction do not have a comma (,) in the correct location.
	<b>ACTION</b>	Inspect the current instruction and change the operands to the instruction so that the are valid.

Diagnostic Messages  
Error Messages

<b>1101</b>	<b>MESSAGE</b>	Standard input must be a pipe or FIFO, not a TTY or device file.
	<b>CAUSE</b>	The assembler was invoked without any input files on the command-line. For this situation the assembler will then attempt to use the standard input of a UNIX pipe command. The object file will be written to standard output for this case. However, before assembling the standard input, the assembler will attempt to discover if the standard input file is associated with a TTY or terminal. If the assembler determines that standard input is associated with a TTY or terminal then it prints this error message and exits.
	<b>ACTION</b>	Supply a filename when invoking the assembler, or use a UNIX pipe to provide an input file for the assembler.
<b>1102</b>	<b>MESSAGE</b>	Displacement of %s must be multiple of four
	<b>CAUSE</b>	The displacement in this instruction must be a multiple of four.
	<b>ACTION</b>	Use a displacement that is a multiple of four.
<b>1103</b>	<b>MESSAGE</b>	Displacement of %s must be zero with ,O
	<b>CAUSE</b>	The displacement must be zero when using the ,O completer.
	<b>ACTION</b>	Use a displacement of zero.
<b>1104</b>	<b>MESSAGE</b>	Displacement can't be zero with ,MA
	<b>CAUSE</b>	The displacement cannot be zero when using the ,MA completer.

<b>1105</b>	<b>ACTION</b>	Use a non-zero displacement.
	<b>MESSAGE</b>	Displacement of %s must be multiple of eight
	<b>CAUSE</b>	The displacement in this instruction must be a multiple of eight.
	<b>ACTION</b>	Use a displacement that is a multiple of eight.

## Panic Messages

The following messages describe panic errors that cause the Assembler to terminate immediately and prevent it from creating an object file. You must correct these errors to assemble your program.

<b>2000</b>	<b>MESSAGE</b>	Exceeded maximum error count
	<b>CAUSE</b>	More than 100 errors were detected and the <code>-e</code> option was not used.
	<b>ACTION</b>	Use the <code>-e</code> option to permit up to a million errors.
<b>2002</b>	<b>MESSAGE</b>	Unable to open input file: %s
	<b>CAUSE</b>	The input file is either nonexistent or unreadable.
	<b>ACTION</b>	Check for presence of requested input file and examine the read permission for the file.
<b>2003</b>	<b>MESSAGE</b>	Unable to open output file: %s
	<b>CAUSE</b>	One of the following: 1. Output file exists and is not writable. 2. Directory is not writable 3. File system is not writable. 4. File system full 5. File system error.
	<b>ACTION</b>	Perform the corresponding action: 1. Delete output file or make output file writable. 2. Make directory writable. 3. Use a read/write file system for the output file. 4. Contact your HP-UX system administrator 5. Contact your HP-UX system administrator
<b>2004</b>	<b>MESSAGE</b>	Free storage exhausted
	<b>CAUSE</b>	Assembler cannot allocate memory for it's internal structures.



	<b>ACTION</b>	Break up the program into smaller modules. If this does not work contact your HP-UX system administrator.
<b>2005</b>	<b>MESSAGE</b>	Internal instruction parsing error on %s
	<b>CAUSE</b>	Assembler has an internal defect.
	<b>ACTION</b>	Have your HP-UX system administrator contact HP Technical Support.
<b>2006</b>	<b>MESSAGE</b>	Unable to regain access to source file for listing
	<b>CAUSE</b>	Not able to access source file for reading, while formatting the assembly listing file.
	<b>ACTION</b>	Check for existence of source file and permission to read it.
<b>2007</b>	<b>MESSAGE</b>	Unable to access temporary file to build listing
	<b>CAUSE</b>	Not able to write to the temporary listing file. Could be a file system error.
	<b>ACTION</b>	Contact your HP-UX system administrator.
<b>2008</b>	<b>MESSAGE</b>	Unterminated macro definition
	<b>CAUSE</b>	Macro definition is not complete until a .ENDM is encountered.
	<b>ACTION</b>	Insert a .ENDM at the end of the macro definition.

## User Warning Messages

The following messages are user warnings. The Assembler will proceed, and produce an object file, in some cases taking the corrective action described.

<b>7000</b>	<b>MESSAGE</b>	Model number is unknown; will default to arch-rev code generation.
	<b>CAUSE</b>	The model number given on a +DA option is not known to the compiler.
	<b>ACTION</b>	The default code generation is as specified in the warning. If this is not the desired target architecture revision, specify the version using an architecture revision (e.g., 1.1) instead of a model number on the +DA option.
<b>7001</b>	<b>MESSAGE</b>	Architecture version is unknown; will default to <i>arch-rev</i> code generation.
	<b>CAUSE</b>	The architecture revision given on a +DA option is not known to the compiler.
	<b>ACTION</b>	The default code generation is as specified in the warning. If this is not what is desired, an alternate architecture revision may be specified.
<b>7002</b>	<b>MESSAGE</b>	Cannot open sched.models. (7002)
	<b>CAUSE</b>	The file sched.models does not exist or cannot be opened for reading.

	<b>ACTION</b>	Check protections on <code>/opt/langtools/lib/sched.models</code> and <code>/usr/lib/sched.models</code> . If neither file exists, contact your HP Service Representative.
<b>7003</b>	<b>MESSAGE</b>	Improper argument to +DA option. (7003)
	<b>CAUSE</b>	An improper argument was given to the +DA or +DS option.
	<b>ACTION</b>	The +DA <i>model</i> is either a model number (such as 877 or I50, or one of the PA-RISC architecture designations 1.0 or 1.1. The +DS <i>model</i> is either a model number (such as 877 or I50), or one of the PA-RISC processor names (such as PA7100). See the <code>/opt/langtools/lib/sched.models</code> file for model numbers, architectures, and processor names.
<b>7004</b>	<b>MESSAGE</b>	Debug information may be corrupt: %1s unresolvable reference %2s (7004)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7005</b>	<b>MESSAGE</b>	Unrecognized opcode %1s (7005)
	<b>CAUSE</b>	The opcode specified in an inline assembly call was invalid.
	<b>ACTION</b>	Check the architecture instruction set specification to determine valid opcode names.
<b>7006</b>	<b>MESSAGE</b>	Improper completer ,%1s given for opcode %2s (7006)

Diagnostic Messages  
User Warning Messages

	<b>CAUSE</b>	The completer specified in an inline assembly call was invalid for the opcode given.
	<b>ACTION</b>	Check the architecture instruction set specification to determine valid completers.
<b>7100</b>	<b>MESSAGE</b>	code subspace has no unwind subspace (7100)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7101</b>	<b>MESSAGE</b>	Improper completer , <i>completer</i> given for opcode <i>opcode</i>
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7102</b>	<b>MESSAGE</b>	Immediate value of <i>value</i> for 5-bit-field in <i>opcode</i> not in [0..31]
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7103</b>	<b>MESSAGE</b>	Extract/deposit of <i>value</i> for field size in <i>opcode</i> not in [1..32]
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7104</b>	<b>MESSAGE</b>	Immediate value of <i>value</i> for <i>opcode</i> is less than -16 (set to -16)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.

<b>7105</b>	<b>MESSAGE</b>	Immediate value of <i>value</i> for <i>opcode</i> is greater than 15 (set to 15)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7106</b>	<b>MESSAGE</b>	DSR value of %1s for %2s not in [0..3] - truncated (7106)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7107</b>	<b>MESSAGE</b>	CSR value of <i>value</i> for <i>opcode</i> not in [0..7] - truncated
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7108</b>	<b>MESSAGE</b>	The value <i>hex-value</i> did not fit into a signed <i>number</i> bit field at offset <i>0xinstruction-offset</i> (op code - <i>op-number</i> )
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7109</b>	<b>MESSAGE</b>	Tried to define value of non-absolute symbol %1s (7109)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7110</b>	<b>MESSAGE</b>	Instruction bypassed low-level manip (7110)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.

Diagnostic Messages  
User Warning Messages

<b>7111</b>	<b>MESSAGE</b>	Bad annotation (7111)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7112</b>	<b>MESSAGE</b>	Mandatory completer missing for opcode %1s (7112)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.

---

## Limit Error Messages

The following messages describe limit errors that cause the Assembler to terminate immediately and prevent it from creating an object file. you may be able to work around these errors. They involve running into Assembler limits or running out of memory.

<b>7200</b>	<b>MESSAGE</b>	<code>start/new_pool: out of memory. (7200)</code>
	<b>CAUSE</b>	The compiler attempted to allocate some dynamic memory, and the system was unable to provide the memory.
	<b>ACTION</b>	<p>The easiest workaround is to break your compilation unit into two or more pieces and compile them separately.</p> <p>On HP-UX, this error may also be produced if the system runs out of swap space. You can increase the amount of swap space available to the system (see your HP-UX system administrator). However, this should only be a last-resort.</p>
<b>7201</b>	<b>MESSAGE</b>	<code>new_slc_block: out of memory. (7201)</code>
	<b>CAUSE</b>	The compiler attempted to allocate some dynamic memory, and the system was unable to provide the memory.
	<b>ACTION</b>	<p>See message 7200. Check the system limits because other processes might be running that also allocate dynamic memory.</p> <p>Break up your compilation module into smaller pieces, and compile them separately.</p>

Diagnostic Messages  
Limit Error Messages

<b>7202</b>	<b>MESSAGE</b>	Increase the system swap area. init_link: out of memory. (7202)
	<b>CAUSE</b>	Compiler ran out of virtual memory. The compiler attempted to allocate some dynamic memory, and the system was unable to provide the memory.
	<b>ACTION</b>	See message 7200. Check the system limits because other processes might be running that also allocate dynamic memory.  Break up your compilation module into smaller pieces, and compile them separately.
<b>7203</b>	<b>MESSAGE</b>	Increase the system swap area. allocate_bytes: out of memory. (7203)
	<b>CAUSE</b>	Compiler ran out of virtual memory. The compiler attempted to allocate some dynamic memory, and the system was unable to provide the memory.
	<b>ACTION</b>	See message 7200. Check the system limits because other processes might be running that also allocate dynamic memory.  Break up your compilation module into smaller pieces, and compile them separately.
<b>7204</b>	<b>MESSAGE</b>	Increase the system swap area. error in writing to output file. (7204)
	<b>CAUSE</b>	I/O error writing to object file.



	<b>ACTION</b>	Check for full file system (HPUX-MPE/iX) or too small object file (MPE/iX).
<b>7205</b>	<b>MESSAGE</b>	unable to allocate space for object in RL. (7205)
	<b>CAUSE</b>	I/O error writing to RL.
	<b>ACTION</b>	Check for too small RL file (MPE/iX).
<b>7206</b>	<b>MESSAGE</b>	unable to add object to RL. (7206)
	<b>CAUSE</b>	I/O error writing to RL.
	<b>ACTION</b>	Check for too small RL file or write permission (MPE/iX).
<b>7207</b>	<b>MESSAGE</b>	object is too big to fit into RL. (7207)
	<b>CAUSE</b>	Object size is too large for the RL requested.
	<b>ACTION</b>	Check for too small RL file or split object up (MPE/iX).
<b>7208</b>	<b>MESSAGE</b>	Internal error while reading %1s (7208)
	<b>CAUSE</b>	An error condition was returned while attempting to open or read data from an object file.
	<b>ACTION</b>	Check status of the object files used to build this program. You might also try recompiling the source file.
<b>7209</b>	<b>MESSAGE</b>	Out of memory while reading %1s (7209)
	<b>CAUSE</b>	Compiler ran out of virtual memory while reading ISOM file.
	<b>ACTION</b>	See message 7200.
<b>7210</b>	<b>MESSAGE</b>	Internal error while writing %1s (7210)

Diagnostic Messages  
Limit Error Messages

	<b>CAUSE</b>	An error condition was returned while attempting to open or write data from an object file.
	<b>ACTION</b>	Check file permissions and the status of object files being written by the compiler.
<b>7211</b>	<b>MESSAGE</b>	Out of memory while writing %1s (7211)
	<b>CAUSE</b>	Compiler ran out of virtual memory while writing ISOM file.
	<b>ACTION</b>	See message 7200.
<b>7212</b>	<b>MESSAGE</b>	get_m_heap: out of memory. (7212)
	<b>CAUSE</b>	Compiler ran out of virtual memory.
	<b>ACTION</b>	See message 7200.
<b>7213</b>	<b>MESSAGE</b>	OUTPUT_byte: out of memory. (7213)
	<b>CAUSE</b>	Compiler ran out of virtual memory.
	<b>ACTION</b>	See message 7200.
<b>7214</b>	<b>MESSAGE</b>	Out of memory while writing ELF file. (7214)
	<b>CAUSE</b>	Compiler ran out of virtual memory.
	<b>ACTION</b>	See message 7200.

---

## Branching Error Messages

The following messages describe branching errors that prevent the Assembler from creating an object file. You must correct these errors to assemble your program.

<b>7400</b>	<b>MESSAGE</b>	Procedure number %1s has no label known to linker (7400)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7401</b>	<b>MESSAGE</b>	Attempt to set location counter backward with .ORIGIN value \ of %1s (7401)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7402</b>	<b>MESSAGE</b>	Procedure call to non entry point: %1s (7402)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7403</b>	<b>MESSAGE</b>	undefined label - %1s (7403)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7404</b>	<b>MESSAGE</b>	branch target %1s unresolvable, instruction number %2s (7404)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.

Diagnostic Messages

Branching Error Messages

<b>7405</b>	<b>MESSAGE</b>	branch target %1s unresolvable, instruction number %2s (7405)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative
<b>7406</b>	<b>MESSAGE</b>	label known to linker deleted (7406)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7407</b>	<b>MESSAGE</b>	Corrupt or unrecognized intermediate code in %1s (7407)
	<b>CAUSE</b>	The ucode in the ISOM file is not recognizable.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7408</b>	<b>MESSAGE</b>	File I/O error while reading %1s (7408)
	<b>CAUSE</b>	A file operation on the ISOM file failed.
	<b>ACTION</b>	Check the reasons for why the file was not readable by user.
<b>7800</b>	<b>MESSAGE</b>	deletion of instruction has removed a target at %1s (7800)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7801</b>	<b>MESSAGE</b>	attempt to delete non-existent instruction (7801)

	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7802</b>	<b>MESSAGE</b>	attempt to insert non-existent inst. (7802)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7203</b>	<b>MESSAGE</b>	attempt to insert labeled instruction (7803)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7804</b>	<b>MESSAGE</b>	set_inst : attempt to set preg field of an instruction (7804)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7805</b>	<b>MESSAGE</b>	internal instruction parsing error (7805)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7806</b>	<b>MESSAGE</b>	re_init_sllic : output file not open (7806)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7807</b>	<b>MESSAGE</b>	re_init_sllic : i/o error (7807)
	<b>CAUSE</b>	Internal compiler error.

Diagnostic Messages  
Branching Error Messages

	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7808</b>	<b>MESSAGE</b>	re_init_sllic : file position out of alignment (7808)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7809</b>	<b>MESSAGE</b>	Data size not equal to subspace length. (7809)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7810</b>	<b>MESSAGE</b>	push_mappings: Stack overflow. (7810)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7811</b>	<b>MESSAGE</b>	pop_mappings: Stack underflow. (7811)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7812</b>	<b>MESSAGE</b>	enter_VT: String too long ( > 8K-12 bytes). (7812)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7813</b>	<b>MESSAGE</b>	fixup_DNTT_entry: no graph entry for symbol %1s. (7813)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.

<b>7814</b>	<b>MESSAGE</b>	fixup_DNTT_entry: can't find procedure end for symbol %1s. (7814)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7815</b>	<b>MESSAGE</b>	Malloc: underflow detected in free(). (7815)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7816</b>	<b>MESSAGE</b>	Malloc: overflow detected in free(). (7816)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7817</b>	<b>MESSAGE</b>	Malloc: Item being freed is not in use. (7817)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7818</b>	<b>MESSAGE</b>	Malloc: Item being freed is of wrong size. (7818)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7819</b>	<b>MESSAGE</b>	Disasm: Attempt to print NIL expression. (7819)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7820</b>	<b>MESSAGE</b>	Disasm: Bad format in format string: %s (7820)

Diagnostic Messages  
Branching Error Messages

	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7821</b>	<b>MESSAGE</b>	after: only one graph entry allowed for repeated inits. (7821)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7822</b>	<b>MESSAGE</b>	newfixup: invalid fixup. (7822)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7823</b>	<b>MESSAGE</b>	xdb_sup: XT entry out of order. (7823)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7824</b>	<b>MESSAGE</b>	inst: Illegal use of pseudo AP register. (7824)
	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7825</b>	<b>MESSAGE</b>	inst: Illegal use of floating point register. (7825)
	<b>CAUSE</b>	Internal compiler error (unless using the Assembler).
	<b>ACTION</b>	Report error to your HP Service Representative.
<b>7826</b>	<b>MESSAGE</b>	Improper format for sched.models. (7826)



	<b>CAUSE</b>	Contents of the file <code>sched.models</code> were unexpected. This file should contain current mappings between machine model numbers and architecture revisions.
	<b>ACTION</b>	Check that <code>/opt/langtools/lib/sched.models</code> or <code>/usr/lib/sched.models</code> were installed or updated along with your current version of the operating system.
<b>7827</b>	<b>MESSAGE</b>	Invalid architecture version. (7827)
	<b>CAUSE</b>	The architecture revision provided in the file <code>sched.models</code> is not known to the compiler.
	<b>ACTION</b>	Check the format of any <code>+DA</code> option specified to the compiler. If none specified or format is correct, check that the file <code>sched.models</code> (in either <code>/opt/langtools/lib</code> or <code>/usr/lib</code> ) was installed or updated along with your current version of the operating system.
<b>7828</b>	<b>MESSAGE</b>	<code>inst: Illegal displacement, low order bits must be zero.</code>
	<b>CAUSE</b>	Internal compiler error (unless using the Assembler).
	<b>ACTION</b>	Report error to your nearest HP Service Representative.
<b>7834</b>	<b>MESSAGE</b>	Procedure limit reached. (7834)
	<b>CAUSE</b>	Too many procedures.
	<b>ACTION</b>	Reduce the number of procedures.
<b>7835</b>	<b>MESSAGE</b>	Internal error encountered while generating ELF file. (7835)

Diagnostic Messages  
Branching Error Messages

	<b>CAUSE</b>	Internal compiler error.
	<b>ACTION</b>	Report error to your HP Service Representative
<b>7836</b>	<b>MESSAGE</b>	Files with executable code must have at least one exported code symbol. (7836)
	<b>CAUSE</b>	Your assembly file contains executable code but does not have any exported entry points.
	<b>ACTION</b>	Use the <code>.EXPORT</code> directive for one of your entry points or code symbols.
<b>7837</b>	<b>MESSAGE</b>	<code>.WORD</code> pseudo-op cannot hold a 64-bit address. Use <code>.DWORD</code> instead. (7837)
	<b>CAUSE</b>	Your PA2.0W assembly file contains a <code>.WORD <i>expr</i></code> pseudo-operation, where <i>expr</i> is an address expression.
	<b>ACTION</b>	Change the pseudo-op to <code>.DWORD <i>expr</i></code>

---

## Index

---

### Symbols

#

label field, 19

\$BSS\$

64-bit mode, 44

\$BSS\$ subspace, 43, 74

\$CODE\$

64-bit mode, 44

\$CODE\$ subspace, 43, 61, 91

\$DATA\$ subspace, 43, 61, 91

\$DEBUG\$ space, 41

\$DLT\$ subspace, 43

\$GLOBAL\$ subspace, 43, 49

\$LIT\$ subspace, 43

\$MILLICODE\$ subspace, 43

\$PLT\$ subspace, 43

\$PRIVATE\$ space, 40, 41, 43, 49, 74

\$\$SHLIB\_DATA\$ subspace, 43

\$\$SHLIB\_INFO\$ subspace, 43

\$\$SHORTBSS\$ subspace, 43, 50

\$\$SHORTDATA\$ subspace, 43, 50

\$TBSS\$ subspace, 50

\$TEXT\$ space, 40, 41, 43

\$UNWIND\$ subspace, 43

(MRP) Millicode Return Pointer, 70

.(period)

special symbol, 21

.ALIGN directive, 42, 57

.ALLOW directive, 58, 93

.BLOCK pseudo-operation, 60, 101

.BLOCKZ pseudo-operation, 60

.bss

64-bit mode, 44

.BSS predefined subspace directive, 116

.BYTE pseudo-operation, 62

.CALL directive, 63, 86

.CALLINFO directive, 48, 67, 81, 102

.CODE directive, 43, 52

.CODE predefined subspace directive, 116

.COMM directive, 74

.COPYRIGHT directive, 75

.DATA directive, 43, 52

.DATA predefined subspace directive, 116

.DOUBLE pseudo-operation, 77

.DWORD pseudo-operation, 78

.END directive, 79

.ENDM directive, 80, 98

.ENTER pseudo-operation, 28, 48, 52, 67, 81, 83, 95, 102

.ENTRY directive, 83

.EQU directive, 19, 21, 25, 84

.EXIT directive, 83, 102

.EXPORT directive, 48, 85

.FIRST predefined subspace directive, 116

.FLOAT pseudo-operation, 88

.GATE predefined subspace directive, 116

.GLOBAL predefined subspace directive, 116

.GNTT predefined subspace directive, 116

.HALF pseudo-operation, 89

.HEADER predefined subspace directive, 116

.HEAP predefined subspace directive, 116

.IMPORT directive, 49, 90

.LABEL directive, 20, 92

.LEAVE pseudo-operation, 28, 48, 52, 67, 81, 83, 95, 102

.LEVEL directive, 58, 93

.LISTOFF directive, 52, 95

.LISTON directive, 52, 95

.LIT predefined subspace directive, 116

.LNTT predefined subspace directive, 117

.LOCCT directive, 46, 97

.MACRO directive, 19, 21, 37, 80, 98

.MILLICODE predefined subspace directive, 117

.ORIGIN directive, 101

.PCB predefined subspace directive, 117

.PROC directive, 48, 102

.PROCEND directive, 48, 102

.REAL predefined subspace directive, 117

.RECOVER predefined subspace directive, 117

.REG directive, 19, 21, 25, 35, 104

.RESERVED predefined subspace directive, 117

.SHLIB\_VERSION directive, 105

.SHORTDATA predefined subspace directive, 117

.SLT predefined subspace directive, 117

.SPACE directive, 41, 43, 106

.SPNUM pseudo-operation, 108

.STACK predefined subspace directive, 117

.STRING pseudo-operation, 109

.STRINGZ pseudo-operation, 109

.SUBSPA directive, 43, 46, 107, 111

.text

64-bit mode, 44

.UNWIND predefined subspace directive, 117

.VERSION directive, 114

.VT predefined subspace directive, 117

.WORD pseudo-operation, 115

---

## Index

---

### Numerics

64-bit environment, 16, 17, 23, 24, 33, 39, 44  
  *See Also* PA-RISC 2.0W  
  .ALLOW directive, 58  
  .CALL directive, 63  
  .CALLINFO directive, 71  
  .EXPORT Directive, 85  
  .EXPORT directive, 85, 86  
  .LEVEL directive, 93  
  .SUBSPA directive, 111  
  Executable and Linking  
    Format, 16, 17, 111  
  memory, 44

### A

ABSOLUTE  
  symbols, 85  
absolute  
  expressions, 16  
  result, 29  
  symbols, 15  
absolute symbols  
  parenthesized subexpressions,  
    34  
access rights attribute and  
  subspaces, 42  
add and branch conditions, 122  
ADDB pseudo-instruction, 120  
ADDIB pseudo-instruction, 120  
address expressions, 16  
addressing  
  long, 40  
  short, 40  
advancing location counter, 101  
  next alignment boundary, 57  
alignment attribute and  
  subspaces, 42  
allowing a label definition, 92  
arg0  
  registers, 136  
arithmetic

  expressions, 29  
  operators, 29  
as  
  using instead of cc command,  
    136  
as command, 124  
  options, 124  
ASCII value  
  initializing reserved storage to,  
    109  
Assembler, 15  
  expressions, 16  
  features, 15  
  invoking, 123  
  list of directives, 53  
  list of pseudo-operations, 55  
  location counters, 16  
  machine language, 15  
  macro processing, 16  
  mnemonic instructions, 15  
  pseudo-operations, 53  
  relocatable object file, 15  
  source file, 15  
  storage allocation, 16  
  subspaces, 16  
  symbol scope, 16  
  symbolic addresses, 15  
  symbolic constants, 15  
assembling your program, 123,  
  136  
assembly language  
  listing, 51  
  procedures, 39  
  programming example  
    binary search for highest bit  
      position, 130  
  C program calling assembly,  
    136  
  C program generating assembly  
    code, 138  
  copying a string, 132  
  dividing a double-word divi-  
    dend, 134

  programming examples, 129  
  programming for HP-UX, 39  
  programs, 15, 39  
assembly statement  
  comments, 19, 20  
  directives, 19  
  instructions, 19  
  label, 19  
  opcode, 19  
  operands, 19, 20  
  pseudo-operations, 19  
assigning an expression value to  
  an identifier, 84

### B

B pseudo-instruction, 120  
bit-wise operators, 29  
blocks of storage, reserving, 60  
branch statement  
  marking, 63  
  procedure call, 63  
branching messages, 187

### C

C compiler  
  dependencies, 136  
  passing arguments to  
    Assembler, 127  
  passing arguments to C  
    preprocessor, 127  
  passing arguments to linker,  
    127  
C language  
  preprocessor (cpp), 128  
  type string, 109  
calling conventions, 47  
catalog  
  message, 141  
cc command  
  dependencies, 136  
  using, 127  
CODE

---

## Index

- 
- symbols, 85
  - COMB pseudo-instruction, 120
  - COMIB pseudo-instruction, 120
  - command
    - as, 124
    - cc, 127
  - comments field, 19, 20
  - compare and branch conditions, 121
  - compiler conventions, 47
  - completers, 35
    - macros, 37
  - condition
    - add and branch, 122
    - compare and branch, 121
  - constants, 21
    - integer, 21
    - parenthesized subexpressions, 34
    - register-type, 25
  - control registers, 24
  - COPY pseudo-instruction, 120
  - counter, location, 97, 101
  - cpp (C preprocessor), 128
  - creating
    - entry/exit code sequences, 67
    - stack unwind descriptors, 67
  - current procedure
    - describing environment of, 67
  - D**
  - DATA
    - symbols, 85
  - declaring
    - a new space, 106
    - a new subspace, 111
    - beginning of macro definition, 98
    - beginning of procedure, 102
    - end of a procedure, 102
    - predefined subspace, 95
  - defining
    - new instructions, with macros, 37
  - delay slot, 119
    - instructions, 119
  - demonstrating the procedure
    - calling convention, 136
  - dependencies, 136
    - cc command, 136
    - pcc\_prefix.s, 136
  - describing environment of
    - current procedure, 67
  - diagnostic messages, 141
  - directives, 53
    - .ALIGN, 42, 57
    - .ALLOW, 58, 93
    - .CALL, 63, 86
    - .CALLINFO, 48, 67, 81, 102
    - .CODE, 43, 52
    - .COMM, 74
    - .COPYRIGHT, 75
    - .DATA, 43, 52
    - .END, 79
    - .ENDM, 80, 98
    - .ENTRY, 83
    - .EQU, 19, 21, 25, 84
    - .EXIT, 83, 102
    - .EXPORT, 48, 85
    - .IMPORT, 49, 90
    - .LABEL, 20, 92
    - .LEVEL, 58, 93
    - .LISTOFF, 52, 95
    - .LISTON, 52, 95
    - .LOCCT, 46, 97
    - .MACRO, 19, 21, 37, 80, 98
    - .ORIGIN, 101
    - .PROC, 48, 102
    - .PROCEND, 48, 102
    - .REG, 19, 21, 25, 35, 104
    - .SHLIB\_VERSION, 105
    - .SPACE, 41, 43, 106
    - .SUBSPA, 43, 46, 107, 111
    - .VERSION, 114
  - list of, 53
  - predefined subspace, 116
  - dp register, 49
  - DS
    - divide step example, 134
  - E**
  - ending the program, 79
  - ENTRY
    - symbols, 85
  - entry points
    - marking procedure, 81, 83
  - error messages, 141, 154
  - errors
    - out of memory, 183
  - examples, 129
    - binary search for highest bit position, 130
    - C program calling assembly, 136
    - C program generating assembly code, 138
    - copying a string, 132
    - dividing a double-word dividend, 134
  - Executable and Linking Format
    - 64-bit environment, 16, 17, 111
  - executable program file, 15
  - exit points
    - marking procedure, 81, 83
  - expansion of macros, 95
  - expressions, 16
    - absolute, 16
    - absolute result, 29
    - address, 16
    - assigning value to an identifier, 84
    - integer constants, 29
    - relocatable, 16
    - relocatable result, 29
    - symbolic addresses, 29
    - symbolic constants, 29

---

## Index

---

- F**  
feature  
  implementation-specific, 58, 93  
field selectors, 30, 51  
  shared libraries, 33  
fields  
  comments, 19, 20  
  label, 19  
  opcode, 19  
  operands, 19, 20  
fixed argument list, 68  
floating-point  
  registers, 23  
floating-point value  
  initializing a double-word to, 77  
  initializing a single-word to, 88  
following instruction  
  delay slot, 119  
frame marker, 68
- G**  
general registers, 23, 35, 48  
generating  
  entry/exit code sequences, 67  
  stack unwind descriptors, 67  
global symbol, 49
- H**  
hard\_reg.h header file, 127  
high-level language procedure, 47  
HP C/HP-UX, 49  
HP FORTRAN 77/HP-UX, 49  
HP Pascal/HP-UX, 49  
  procedures, 48
- I**  
identifier, assigning an  
  expression value to, 84
- illegal symbols, 21  
implementation-specific  
  features, 58, 93  
initializing  
  block of storage, 60, 62, 89  
  double-word to floating-point  
  value, 77  
  reserved storage, 78, 115  
  reserved storage to ASCII  
  values, 109  
  single-word to floating-point  
  value, 88  
inserting  
  copyright notice, 75  
  version string, 114  
instruction set, 119  
instructions  
  creating with macros, 37  
  delay slot, 119  
  pseudo-instruction, 120  
integer constants, 21  
invoking the Assembler, 123
- L**  
label definition, permitting, 92  
label field, 19  
  .EQU, 19  
  .MACRO, 19  
  .REG, 19  
  pound sign (#), 19  
ld(1), 42  
LDI pseudo-instruction, 120  
legal combination  
  relocatable terms, 30  
legal symbols, 21  
levels, versions of PA-RISC, 58, 93  
limit messages, 183  
limits  
  memory, 183  
linker, 15  
  executable program file, 15  
  ld(1), 42  
  program file, 15  
  relocatable object file, 15  
  *See also* Executable and  
  Linking Format  
  subspaces, 42  
listing  
  assembly, 51  
location counters, 46, 62, 97  
  advanced, 101  
  local to Assembler, 46  
  next alignment boundary, 57  
  long addressing, 40
- M**  
macros, 37  
  .ENDM directive, 80  
  .MACRO directive, 98  
  completers, 37  
  creating instructions, 37  
  declaring, 98  
  defining new instructions, 37  
  expansion, 95  
  opcodes, 37  
  operands, 37  
  processing, 16, 37  
  subopcodes, 37  
making  
  a new space, 106  
  entry/exit code sequences, 67  
  stack unwind descriptors, 67  
making symbols available to  
  other modules, 85, 90  
marking  
  beginning of macro, 98  
  beginning of procedure, 102  
  end of macro, 80  
  end of procedure, 102  
  next branch statement, 63  
  procedure entry points, 81, 83  
  procedure exit points, 81, 83  
memory

---

## Index

---

- 64-bit environment, 44
  - See also* storage
  - unable to allocate, 183
- message catalog, 141
- messages
  - branching, 187
  - limit, 183
  - out of memory, 183
  - user warnings, 178
  - warning warnings, 187
- MFCTL
  - thread local storage, 50
- millicode, 70
- Millicode Return Pointer (MRP), 70
- mnemonic
  - instructions, 15
  - register, 23
- moving location counter
  - to next alignment boundary, 57
- MTSAR pseudo-instruction, 120
- N**
- new
  - instructions
    - creating with macros, 37
    - subspaces, 106, 111
- NOP pseudo-instruction, 120
- O**
- object file
  - specifying version, 105
- opcode
  - field, 19
  - macros, 37
- operands, 35
  - field, 19, 20
  - macros, 37
- operators, 29
  - arithmetic, 29
  - bit-wise, 29
  - field selectors, 30
- options
  - as command, 124
- P**
- page size, 33
- panic messages, 176
- parameters
  - as command, 124
- parenthesized subexpressions, 16, 34
  - absolute symbols, 34
  - constants, 34
- PA-RISC
  - See Also* 64-bit environment
  - instruction set, 119
  - version levels, 58, 93
- PA-RISC 2.0W, 16, 17, 23, 24, 33, 39, 44
  - See Also* 64-bit environment
- passing Assembler arguments
  - from C compiler, 127
- pcc\_prefix.s, 136
- pcc\_prefix.s configuration file, 127
  - hard\_reg.h, 127
  - soft\_reg.h, 127
  - std\_space.h, 127
- period (.), 21
- permitting a label definition, 92
- PIC (position-independent code), 33, 51
- placing copyright notice, 75
- position-independent code, 33, 51
- pound sign (#), 19
- predefined subspace
  - declarations, 95
- predefined subspace directive, 116
  - .BSS, 116
  - .CODE, 116
  - .DATA, 116
  - .FIRST, 116
  - .GATE, 116
  - .GLOBAL, 116
  - .GNTT, 116
  - .HEADER, 116
  - .HEAP, 116
  - .LIT, 116
  - .LNTT, 117
  - .MILLICODE, 117
  - .PCB, 117
  - .REAL, 117
  - .RECOVER, 117
  - .RESERVED, 117
  - .SHORTDATA, 117
  - .SLT, 117
  - .STACK, 117
  - .UNWIND, 117
  - .VT, 117
- previous\_sp special register
  - mnemonic, 28
- PRI\_PROG
  - symbols, 86
- procedure calling conventions, 47
  - demonstrating, 136
  - registers, 28
- procedures
  - declaring, 102
  - ending, 102
  - marking entry points, 81, 83
  - marking exit points, 81, 83
- processing
  - macros, 16, 37
- programming
  - aids, 116
  - for HP-UX, 39
- programming examples, 129
  - binary search for highest bit position, 130
  - C program calling assembly, 136

---

## Index

---

- C program generating  
  assembly code, 138  
  copying a string, 132  
  dividing a double-word  
    dividend, 134  
programs  
  file, 15  
  structure, 19  
pseudo-instruction, 120  
  ADDB, 120  
  ADDIB, 120  
  B, 120  
  COMB, 120  
  COMIB, 120  
  COPY, 120  
  LDI, 120  
  MTSAR, 120  
  NOP, 120  
pseudo-instruction set, 119  
pseudo-operation, 53  
  .BLOCK, 60, 101  
  .BLOCKZ, 60  
  .BYTE, 62  
  .DOUBLE, 77  
  .DWORD, 78  
  .ENTER, 28, 48, 52, 67, 81,  
    83, 95, 102  
  .FLOAT, 88  
  .HALF, 89  
  .LEAVE, 28, 48, 52, 67, 81, 83,  
    95, 102  
  .SPNUM, 108  
  .STRING, 109  
  .STRINGZ, 109  
  .WORD, 115  
  list of, 55
- Q**  
quadrant attribute and  
  subspaces, 42
- R**  
register  
  control, 24  
  floating-point, 23  
  general, 23, 35, 48  
  mnemonics, 23  
  name user-defined, 104  
  procedure calling convention,  
    28  
  space, 24, 27  
  typing, 25, 35  
register mnemonic  
  previous\_sp, 28  
registers  
  arg0, 136  
  r%26, 136  
  r%28, 136  
  ret0, 136  
relocatable  
  expressions, 16  
  legal combinations, 30  
  result, 29  
  symbols, 15  
relocatable object file, 15  
requesting storage, 74  
reserving a single-word, 108  
reserving storage, 60, 78, 115  
  and initializing to specified  
    value, 62  
  initializing to ASCII values,  
    109  
  initializing to specified value,  
    89  
result  
  absolute, 29  
  relocatable, 29  
ret0  
  registers, 136  
returning  
  to old space, 106  
  to old subspace, 111
- S**  
SEC\_PROG  
  symbols, 86  
sections  
  64-bit mode, 44  
segments  
  64-bit environment, 44  
shared libraries  
  creating, 51  
  field selectors, 33  
  spaces, 40  
  specifying object file version,  
    105  
shared memory  
  spaces, 40  
SHN\_ABS  
  ., 85  
short addressing, 40  
single-word to floating-point  
  value, 88  
soft\_reg.h header file, 127  
sort keys  
  and standard subspaces, 43  
  attribute and subspaces, 42, 43  
  spaces, 41  
source file, 15  
space number, initialized with,  
  108  
spaces  
  \$DEBUG\$, 41  
  \$PRIVATE\$, 40, 41, 43, 49, 74  
  \$TEXT\$, 40, 41, 43  
  .SPACE directive, 41, 106  
  64-bit environment, 44  
  code, 39  
  data, 39  
  declaring, 106  
  declaring new, 106  
  description, 39  
  identifiers, 39  
  memory layout on HP-UX, 41  
  offsets, 40  
  quadrant, 40



---

## Index

---

- registers, 24, 27, 39
- returning to, 106
- shared libraries, 40
- shared memory, 40
- sort key, 41
- unloadable, 41
- special symbol
  - period (.), 21
- specifying
  - end of a macro definition, 80
  - end of a procedure, 102
  - end of a program, 79
  - macro definition, 98
  - new space, 106
  - new subspace, 111
  - next branch statement, 63
  - object file version, 105
  - procedure, 102
  - procedure entry points, 81, 83
  - procedure exit points, 81, 83
- stack
  - fixed argument list, 68
  - frame, 69
  - frame marker, 68
  - unwind, 47
- standard arithmetic operators, 29
- standard procedure calling
  - conventions, 47
- standard subspaces and sort keys, 43
- start/new\_pool
  - out of memory, 183
- statement
  - directives, 19
  - instructions, 19
  - pseudo-operations, 19
- std\_space.h header file, 127
- storage
  - allocation, 16
  - initializing, 78, 115
  - request, 74
  - reserving blocks, 60
- See also* memory
- thread local, 50, 90, 91, 107, 112
- subexpression
  - parenthesized, 16, 34
- subopcode
  - macros, 37
- subspace, 42
  - \$BSS\$, 43, 74
  - \$CODE\$, 43, 61, 91
  - \$DATAS\$, 43, 61, 91
  - \$DLTS\$, 43
  - \$GLOBALS\$, 43, 49
  - \$LITS\$, 43
  - \$MILLICODE\$, 43
  - \$PLTS\$, 43
  - \$SHLIB\_DATAS\$, 43
  - \$SHLIB\_INFO\$, 43
  - \$SHORTBSS\$, 43, 50
  - \$SHORTDATAS\$, 43, 50
  - \$TBSS\$, 50
  - \$UNWIND\$, 43
  - 64-bit environment, 44
  - access rights attribute, 42
  - alignment attribute, 42
  - and location counters, 16
  - attributes, 42
  - declaring, 111
  - linker, 42
  - location counters, 46
  - predefined declarations, 95
  - quadrant attribute, 42
  - returning to, 111
  - sort key attribute, 42, 43
- subspace attribute
  - access rights, 42
  - alignment, 42
  - quadrant, 42
  - sort key, 42, 43
- subspaces
  - declaring new, 111
- swap space
  - errors, 183
- switching
  - to old space, 106
  - to old subspace, 111
- symbolic
  - addresses, 15
  - constants, 15
- symbols, 21
  - ABSOLUTE, 85
  - absolute, 15
  - case sensitive, 48
  - CODE, 85
  - DATA, 85
  - ENTRY, 85
  - exported, 16
  - illegal, 21
  - imported, 16
  - legal, 21
  - period (.), 21
  - PRI\_PROG, 86
  - relocatable, 15
  - scope, 16
  - SEC\_PROG, 86
  - type, 48
  - valid, 21
- symbols available to other modules, 85, 90
- T**
  - terminating the program, 79
  - thread local storage, 50, 90, 91, 107, 112
  - MFCTL, 50
  - TSPECIFIC, 90, 91, 107, 112
- typing
  - register, 35
- U**
  - unloadable space, 41
  - unwind descriptors, 47
  - user warning messages, 178
  - user-defined register name, 104
  - using as command, 124

---

# Index

---

options, 124

## V

valid symbols, 21

version

  .SHLIB\_VERSION directive,  
  105

  in object file, 105

  inserting string, 114

version levels of PA-RISC, 58, 93

virtual address, 39

## W

warnings

  messages, 142, 178