HP 9000
Series 300/400
Computers

C Programmer's Guide

C Programmer's Guide

HP 9000 Series 300/400 Computers

HP Part No. B1864-90008
Printed in USA 01/91

First Edition
E0191

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 manual, 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 direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Copyright © Hewlett-Packard Company 1987, 1988, 1989, 1990, 1991

This document contains information which is protected by copyright. All rights are reserved. 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 Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-7013.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright © AT&T, Inc. 1980, 1984, 1986

Copyright © The Regents of the University of California 1979, 1980, 1983, 1985

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

January 1991 ... Edition 1.

This edition of the *C Programmer's Guide* includes information on shared libraries and a new compiler option for optimizing programs that contain non-reducible flow graphs in them.

As of the 7.40 release, the Series 300/400 compilers generate a different format for debugger information that is incompatible with debugger information generated by previous releases of the compilers. Therefore, if you compile with the `-g` compile-line option, you will not be able to debug this code with pre-7.40 debuggers. Similarly, the 7.40 and later debuggers cannot be used to debug code produced by pre-7.40 Series 300/400 compilers.

Contents

1. Overview

Typographical Conventions 1-2
Compiling a C Program 1-2
Related Documents 1-3

2. C Data Types and Alignments

C Data Types (Sizes and Ranges) 2-2
Data Type Alignments 2-4
Array Size and Alignment 2-8
Alignment within Structures 2-9
An HPUX_WORD Example 2-10
An HPUX_NATURAL Example 2-12
Aligning Structures between Architectures 2-14

3. Optimizing C Programs

The Levels of Optimization 3-2
Invoking Optimization 3-3
Using Directives to Control Optimization 3-5
OPTIMIZE 3-5
OPT_LEVEL 3-6
HP_INLINE_LINES 3-6
HP_INLINE_FORCE 3-7
HP_INLINE_OMIT 3-8
HP_INLINE_DEFAULT 3-9
HP_INLINE_NOCODE 3-10
NO_SIDE_EFFECTS 3-10
Error Messages 3-14
Levels 2 and 3 Optimization Errors 3-14
Errors Caused by Pre-6.5 Compilers 3-16

What to Do About Slow Compilation and Out-of-Memory Conditions . . . 3-17
Troubleshooting Optimization Problems . . . 3-19
A Closer Look at Optimizations . . . 3-20
Peephole Optimization . . . 3-21
Instruction Scheduling . . . 3-22
Constant Folding . . . 3-23
Constant Propagation . . . 3-23
Dead Code Elimination . . . 3-24
Coloring Register Allocation . . . 3-24
Common Subexpression Elimination . . . 3-25
Dead Store Elimination . . . 3-26
Copy Propagation . . . 3-26
Loop Unrolling . . . 3-27
Code Motion of Loop Invariants . . . 3-28
Strength Reduction of Loop Induction Variables . . . 3-30
Elimination of Tail Recursion . . . 3-31
Non-Reducible Code . . . 3-32
Procedure Integration . . . 3-34

4. Implementation Dependencies
Implementation Dependencies for C 7.0 . . . 4-1
Primary Name Definitions in C Libraries . . . 4-1
Implementation Dependencies for C 8.0 . . . 4-2
Support for Shared Libraries . . . 4-2

5. Porting to ANSI C
The `const` and `volatile` Qualifiers . . . 5-1
The const Qualifier . . . 5-2
The volatile Qualifier . . . 5-3
Upgrading Existing C Programs to Use Prototypes . . . 5-4
Advantages of the Function Prototype . . . 5-4
Function Prototype Considerations . . . 5-6
How the Name Spaces Work for ANSI C and Other Standards . 5-11
HP Header File and Library Implementation of Name Space . 5-12
Silent Changes for ANSI C . . . 5-14

A. Implementation-Defined Behaviors and Extensions to ANSI-C
Implementation-Defined Behaviors A-1
Diagnostic Messages A-1
Arguments to main () A-2
Interactive Device A-2
Identifiers A-2
Handling Characters A-3
Handling Characters (continued) A-4
Handling Integers A-4
Handling Floating-Point Values A-5
Handling Arrays and Pointers A-6
Registers A-6
Handling of Structures, Unions, Enumerations and Bit Fields . A-7
Qualifiers A-7
Declarator Limits A-8
Case Limits A-8
Preprocessing Directives A-9
Library Functions A-10
Library Functions (continued) A-12
Library Functions (continued) A-14
Implementation-Defined Extensions A-15

B. HP-UX Reference Pages

Index

Figures

2-1. Comparison between HP-UX and DOMAIN Bit Alignments . 2-7
2-2. Structure Definition for Illustrating Storage and Alignment . . 2-9
2-3. How the Data is Stored in Memory Using the HPUX_WORD Alignment Mode 2-10
2-4. How the Data is Stored in Memory Using the HPUX_NATURAL Alignment Mode 2-12
2-5. Comparison between HPUX_WORD and HPUX_NATURAL Byte Alignments 2-15
3-1. Example of Using NO_SIDE_EFFECTS 3-13
3-2. Non-Reducible Flow Graph 3-33

Tables

2-1. C Data Types for Series 300/400 Computers 2-2
2-2. Comparison of Alignment Rules 2-4
2-3. Description of Padding 2-11
2-4. Padding in HPUX_NATURAL Mode 2-13
3-1. C Optimization Levels 3-2
3-2. Optimization Compiler Options 3-4
5-1. Selecting a Name Space in ANSI Mode 5-13

1

Overview

This *Guide* describes some essential elements of using HP's implementation of the C and ANSI C language on Series 300/400 computers. Specifically, it contains the following chapters:

- Chapter 1: Overview

  Gives an overview of C on Series 300/400 computers; directs you to related, useful C documentation; and lists the typographical conventions used throughout the book.

- Chapter 2: C Data Types

  Describes C data types as implemented on Series 300/400 computers. To aid programmers in porting code between systems, this chapter also includes information on Series 500, Series 600/700/800 and HP Apollo C data types.

- Chapter 3: Optimizing C Programs

  Discusses how to optimize C programs using compiler options and directives. Also discusses possible incompatibilities that may occur when linking object files created by the C compiler in release 6.5 and later with object files created by compilers from previous (pre-6.5) compilers.

- Chapter 4: Implementation Dependencies

  Describes any implementation dependencies of a new revision.

- Chapter 5: Porting to ANSI C

  Describes the process of moving existing programs to ANSI C. It also describes the type qualifiers `const` and `volatile`.

- Appendix A: Implementation-Defined Behaviors and Extensions to ANSI C

  Provides information on all implementation-defined behaviors and extensions for the Series 300/400 ANSI C product.

This book does not, however, teach the C language or provide detailed C reference information. Such detail is provided in *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr., a companion to this document.

Typographical Conventions

The following conventions are used throughout this manual:

- C programs and examples appear in `computer font`.
- References of the form "*name(N)*" refer to a command, system call, or library routine in the *HP-UX Reference*. For example, *cc(1)* refers to the `cc` page in section 1 of the *HP-UX Reference*.
- User-supplied information appears in *italic font*. For example:

  `cc` *file_name*

  This means you would type `cc` followed by a file name of your choice.

Compiling a C Program

To compile a C program, use either the *c89(1)* or the *cc(1)* command which are covered in the *HP-UX Reference* manual.

The syntax used to compile a C program is as follows:

`cc` *file_name*

where *file_name* is the C source file to be compiled. Executing this command, produces an executable file named `a.out`.

The syntax used to compile an ANSI C program is as follows:

```
cc -Aa file_name
```

or

```
c89 file_name
```

where `-Aa` is the option for compiling an ANSI C program and *file_name* is the ANSI C source file to be compiled. The `c89` command is the POSIX command for compiling with ANSI Standard C.

Related Documents

In addition to this book and *C: A Reference Manual*, you may find the following books useful:

- *HP-UX Reference*—provides detailed information for HP-UX commands, system calls to the kernel, standard and non-standard library routines, file formats, and device drivers.
- *HP-UX Portability Guide*—assists programmers in porting from one HP-UX language to another, or from non-HP-UX systems to HP-UX and vice versa.
- *Programming on HP-UX*—describes programming in general on HP-UX. For example, it covers linking, loading, and several other HP-UX programming features.
- *HP-UX Documentation Roadmap*—summarizes the entire HP-UX documentation set and cross-references manuals by topics.
- *American National Standard for Information Systems—Programming Language C, ANS X3.159-1989*—provides detailed reference information on the C programming language.

2

C Data Types and Alignments

This chapter summarizes the C data types. Specifically, it describes:

- data types, their sizes, and their ranges
- data type alignments
- array size and alignment
- structure size and alignment.

To aid you in porting programs, this chapter also provides relevant information for C data types on Series 500, Series 600/700/800, and HP Apollo computers.

C Data Types (Sizes and Ranges)

Table 2-1 lists the Series 300/400 C data types, their sizes, and range of possible values.

Table 2-1. C Data Types for Series 300/400 Computers

Data Type	Size	Range
`char` `signed char`[1]	1 byte	−128 to 127
`unsigned char`	1 byte	0 to 255
`short` `signed short`[1]	2 bytes	−32768 to 32767
`unsigned short`	2 bytes	0 to 65535
`int` `signed`[1] `signed int`[1]	4 bytes	-2^{31} to $2^{31} - 1$
`unsigned int` `unsigned`	4 bytes	0 to $2^{32} - 1$
`long` `signed long`[1]	4 bytes	same as **`int`**
`unsigned long`	4 bytes	same as **`unsigned int`**
`float`	4 bytes	−3.402 823E+38 to −1.175 495E−38 0.0 1.175 495E−38 to 3.402 823E+38
`double`	8 bytes	−1.797 693 134 862 31D+308 to −2.225 073 858 507 21D−308 0.0 2.225 073 858 507 21D−308 to 1.797 693 134 862 31D+308

**Table 2-1.
C Data Types for Series 300/400 Computers (continued)**

Data Type	Size	Range
`enum`	4 bytes	2^{32} different enumerated values (from 0 to $2^{32} - 1$) (but limited by compiler's symbol table size)
`pointer`	4 bytes	a 32-bit address
`long double`[2]	16 bytes	-1.18973149535723176508575932662800070162E4932 to -3.36210314311209350626267781732175260026E-4932 0.0 3.36210314311209350626267781732175260026E-4932 to 1.18973149535723176508575932662800070162E4932

1 The `signed` keyword is not available on Series 500.

2 The `long double` type is only available in ANSI C. Note that ANSI C is not available on Series 500 computers.

Data Type Alignments

The data alignment mode provided on the various HP and HP Apollo systems can be divided into two main categories, *WORD* and *NATURAL. WORD* alignment is typical of the Series 300/400-based systems and is so called because the largest alignment boundary is a word (i.e. 2 bytes). *NATURAL* alignment is typical of RISC-based systems and is characterized by the alignment of an object being relative to the size of the object.

On HP systems, there are three distinct alignments:

HPUX_WORD	The native alignment for Series 300/400
HPUX_NATURAL_S500	The native alignment for HP Series 500
HPUX_NATURAL	The native alignment for Series 600/700/800

On HP Apollo systems, there are two distinct alignments:

DOMAIN_WORD	The native alignment for HP Apollo Series 3000/4000
DOMAIN_NATURAL	The native alignment for HP Apollo Series 10000

Finally there is an architecture-independent alignment mode called NATURAL. The purpose of NATURAL is to provide a common alignment mode across all HP and HP Apollo systems. NATURAL is the recommended alignment mode to be used when writing programs that share data among several types of HP and HP Apollo systems.

The following table illustrates the differences between the various alignment modes.

Table 2-2. Comparison of Alignment Rules

Type	Size	HPUX_WORD & DOMAIN_WORD	HPUX_NATURAL & DOMAIN_NATURAL	HPUX_ NATURAL_S500	NATURAL
char signed char unsigned char	1 byte	1 byte	1 byte	1 byte	1 byte
short signed short unsigned short	2 bytes	2 byte	2 byte	2 byte	2 byte

Table 2-2. Comparison of Alignment Rules (continued)

Type	Size	HPUX_WORD & DOMAIN_WORD	HPUX_NATURAL & DOMAIN_NATURAL	HPUX_ NATURAL_S500	NATURAL

`int`
`signed int`
`unsigned int`
`long`
`signed long`
`unsigned long`
	4 bytes	4 byte[1]	4 byte	4 byte	4 byte

`float`	4 bytes	4 byte[1]	4 byte	4 byte	4 byte
`double`	8 bytes	4 byte[1]	8 byte	4 byte	8 byte
`long double`[4]	16 bytes or 8 bytes	4 byte[1]	8 byte	n.a.	8 byte
`enum`	4 bytes	4 byte[1]	4 byte	4 byte	4 byte
arrays	Same as the alignment rule for the type of the array element.				
`struct`	[2,3]	2 byte[1]	1-, 2-, 4- or 8 byte[2,3]	2- or 4 byte[2,3,5]	2-, 4- or 8 byte[2,3]
`union`	[2,3]	2 byte[1]	1-, 2-, 4- or 8 byte[2,3]	2- or 4 byte[2,3,5]	2-, 4- or 8 byte[2,3]
bit fields	Same alignment as declared type.				

1 Within a structure on `HPUX_WORD`, all types larger than 2 bytes are aligned on a 2-byte boundary.

2 Same as the alignment of the largest member.

3 Padding is done to a multiple of the alignment size.

4 The `long double` type is only available in ANSI C and on HP-UX alignment modes (as well as `NATURAL`), its size is 16 bytes. On DOMAIN alignment modes, `long double` is treated as `double` (i.e., size 8 bytes).

5 Series 500 computers align structures on 2 or 4 byte boundaries.

2

With the exception of bit fields, `DOMAIN_WORD` structure alignment is the same as for Series 300/400 computers and `DOMAIN_NATURAL` structure alignment is the same as for Series 600/700/800 computers.

HP-UX and DOMAIN use different algorithms for aligning bit fields within a structure. HP-UX aligns a bit field *f* within a structure by taking the modulo (%) of the *current_offset* prior to *f* within the structure and the *alignment_of_f* and adding this result to the *width_of_f*. The syntax for this algorithm is:

```
(current_offset % alignment_of_f) + width_of_f
```

If this exceeds the size of *f*, then it aligns *f* according to its type; otherwise, it allocates it at the offset prior to *f*. DOMAIN uses a similar algorithm to HP-UX's for aligning bit fields within a structure; however, all of its bit fields are treated as type `int`, regardless of their declared type. For example, consider the following `struct` statement:

```
struct S {char a:4; char b:3; char c:2;}
```

Under the HP-UX scheme, `a` is at bit offset 0, `b` is at bit offset 4, and `c` is at bit offset 8, not at the next bit 7. The reason is bit alignment follows this algorithm:

```
(7 % 8) + 2
```

which results in a value of 9. This exceeds the size of type `char` (8 bits). So, field `c` is aligned according to its type (that is, on the next byte boundary).

The DOMAIN algorithm for aligning bit fields is the same as HP-UX's, except all bit fields are treated as type `int`, regardless of their user declared type. The previous example would look like this using the DOMAIN algorithm:

```
(7 % 16) + 2
```

which results in a value of 9. This does not exceed the size of type `int` (32 bits). So, field `c` is aligned at bit offset 7 as seen in the following figure.

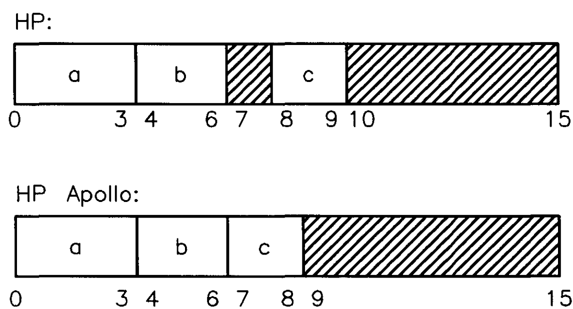


Figure 2-1. Comparison between HP-UX and DOMAIN Bit Alignments

With respect to size, DOMAIN treats all bit fields as `int`. Therefore, the following `struct` is allowed:

```
struct Group_members {
       char letter:10;
       int  number;
       } gp_mem;
```

This same `struct` declaration would cause an error under HP-UX alignment rules since 10 bits exceeds the size of the data type `char`.

`NATURAL` uses the same scheme for bit fields as DOMAIN.

Array Size and Alignment

An array's size is computed as:

(size of array element type) × (number of elements)

For instance, the array declared below is 400 bytes (4 × 100) long:

```
int     arr[100];
```

The size of the array element type is 4 bytes and the number of elements is 100.

On HPUX_WORD, arrays are aligned on a boundary corresponding to their element type. For example, a `double` array would be aligned on a 4-byte boundary; and a `float` array within a `struct` would be aligned on a 2-byte boundary.

Alignment within Structures

In a structure, each member is allocated sequentially at the next alignment boundary corresponding to its type. Therefore, the structure might be padded if its members' types have different alignment requirements. In a union, all members are allocated starting at the same memory location. Figure 2-2 defines a structure we will use to illustrate alignment within structures.

```
struct x {
        char        y[3];
        short       z;
};
struct q {
        char        n;
        struct      x v[2];
        double      u;
        char        t;
        int         s:6;
        char        m;
        long double g;
} a = {1,2,3,4,5,6,7,8,9,10,11,12,13,14};
```

Figure 2-2. Structure Definition for Illustrating Storage and Alignment

An HPUX_WORD Example

Figure 2-3 shows how the data in Figure 2-2 is stored in memory. The values are shown in parentheses above the structure member names. Memory locations containing slashes are *padding* or *filler* bytes. Note that in the HPUX_WORD (and HPUX_NATURAL_S500 mode) mode a structure will be padded at the end if needed to make its size a multiple of 2 bytes.

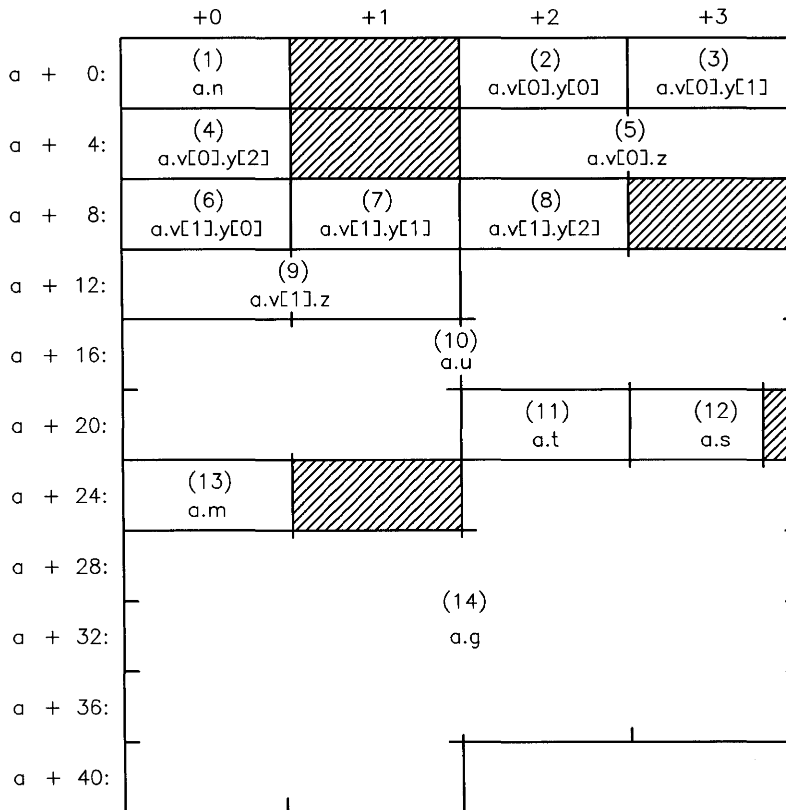


Figure 2-3.
How the Data is Stored in Memory Using the HPUX_WORD Alignment Mode

The following table summarizes why padding occurred where it did in Figure 2-3.

Table 2-3. Description of Padding

Padding	Reason for Padding
`a + 1`	The most restrictive type of `struct x` is `short`; therefore, the structure is 2-byte aligned.
`a + 5`	Aligns the `short z` on a 2-byte boundary.
`a + 11`	Aligns the `short z` on a 2-byte boundary.
`a + 23`	Aligns the `char m` to a 1-byte boundary.
`a + 25`	Aligns the `long double g` on a 2-byte boundary.

An HPUX_NATURAL Example

Figure 2-4 shows how the members of the structure defined in Figure 2-2 would be aligned in memory in the HPUX_NATURAL mode. The structure itself starts on an 8-byte boundary because this is the largest type within the structure.

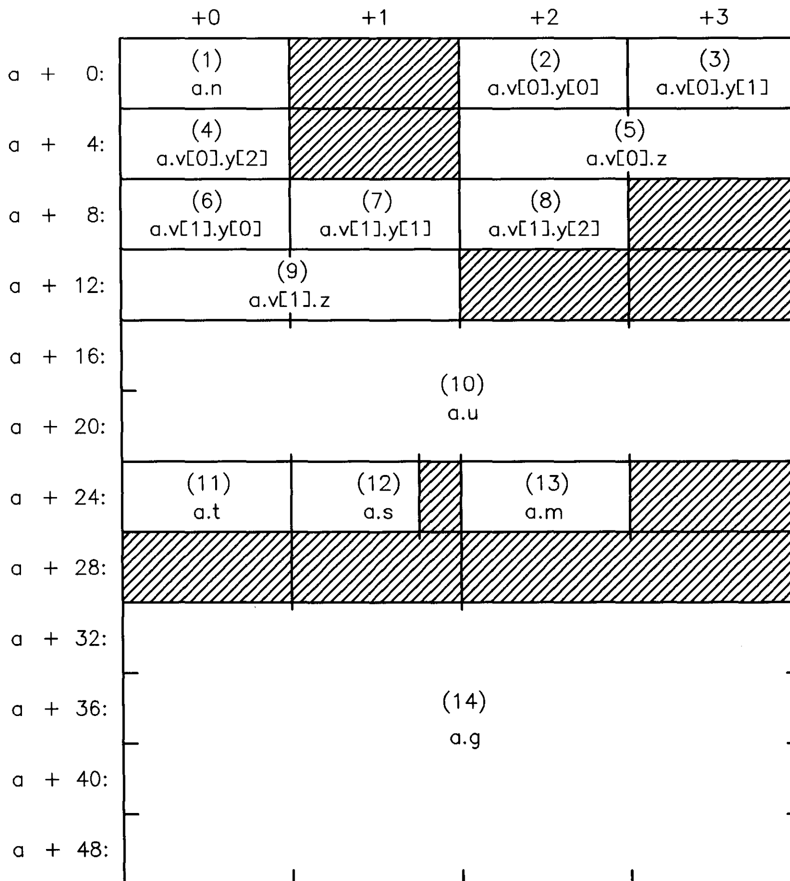


Figure 2-4.
How the Data is Stored in Memory Using the HPUX_NATURAL Alignment Mode

Table 2-4 describes why the padding occurs where it does in this structure.

Table 2-4. Padding in HPUX_NATURAL Mode

Padding	Reason for Padding
`a + 1`	The largest type of the structure `x` is `short`; therefore, the structure is 2-byte aligned.
`a + 5`	Aligns the `short z` on a 2-byte boundary.
`a + 11`	Aligns the `short z` on a 2-byte boundary.
`a + 14` through `a + 15`	Aligns the `double u` on an 8-byte boundary.
`a + 25`	Aligns the `char m` to a 1-byte boundary.
`a + 27` through `a + 31`	Aligns the `long double g` on an 8-byte boundary.

Aligning Structures between Architectures

The purpose of the HP_ALIGN pragma is to provide HPUX_NATURAL_S500, HPUX_NATURAL, DOMAIN_WORD, and DOMAIN_NATURAL alignment on computers that use HPUX_WORD alignment. This provides a measure of portability between architectures. To illustrate the portability problem raised by different alignments, consider the following example.

```
#include <stdio.h>

struct char_int
  {
    char field1;
    int  field2;
  };

main ()
  {
  FILE *fp;
  struct char_int s;
                .
                .
                .
  fp = fopen("myfile", "w");
  fwrite(&s, sizeof(s), 1, fp);
                .
                .
                .
  }
```

The alignment for the struct that is written to myfile in the above example is shown in the following diagram.

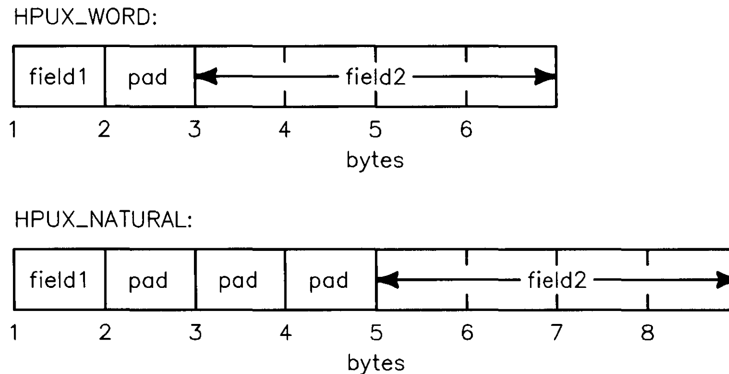


Figure 2-5.
Comparison between HPUX_WORD and HPUX_NATURAL Byte Alignments

In the HPUX_WORD alignment mode, this results in 6 bytes being written to the myfile. The above integer field2 begins on the 3rd byte. Whereas, in the HPUX_NATURAL alignment mode the previous program segment results in 8 bytes being written to myfile. The integer field2 begins on the 5th byte.

The HP_ALIGN pragma

The syntax for this pragma is:

```
#pragma HP_ALIGN align_mode [PUSH]

#pragma HP_ALIGN POP
```

where *align_mode* is one of the following:

HPUX_WORD	Series 300/400 computer's alignment mode (default)
HPUX_NATURAL_S500	Series 500 computer's alignment mode
HPUX_NATURAL	Series 600/700/800 computer's alignment mode
NATURAL	provides a consistent alignment scheme across HP architectures
DOMAIN_WORD	word alignment mode on HP Apollo architecture
DOMAIN_NATURAL	natural alignment mode on HP Apollo architecture

If the optional parameter PUSH is specified with an *align_mode*, the current alignment mode is saved (on an alignment mode stack) and the specified *align_mode* becomes the new alignment mode.

The second syntax (#pragma HP_ALIGN POP) restores the alignment mode that was last pushed on the alignment mode stack. If the alignment mode stack is empty, the compiler issues a diagnostic message.

The pragma must have a global scope (i.e., outside of any function or enclosing struct or union). For example, given the following sequence of pragmas:

```
#pragma HP_ALIGN HPUX_WORD PUSH

struct string_1 {
        char *c_string;
        int  counter;
       };

#pragma HP_ALIGN HPUX_NATURAL PUSH

struct car {
        long double  car_speed;
        char         *car_type;
       };

#pragma HP_ALIGN POP

struct bus {
        int bus_number;
        char bus_color;
       };

#pragma HP_ALIGN POP
```

Any variables declared of type struct string_1, would be aligned according to the HPUX_WORD alignment mode; any variables declared of type struct car, would be aligned according to the HPUX_NATURAL alignment mode; any variables declared of type struct bus would be aligned according to HPUX_WORD.

3

Optimizing C Programs

This chapter describes how to use the optimization capabilities of the C compiler. In particular, this chapter discusses:

- an overview of the levels of optimization
- how to invoke optimization
- using compiler directives to control optimization
- messages produced by the compiler during optimization
- troubleshooting optimization problems
- the actual code transformations performed during optimization.

The Levels of Optimization

The C compiler provides four levels of optimization: levels 0, 1, 2, and 3. Table 3-1 summarizes each level, gives the advantages and disadvantages of using the level, and recommends when to use the level.

Table 3-1. C Optimization Levels

Level	Optimizations Performed	Advantages	Disadvantages
0	Constant folding simple register assignment.	Compiles fastest. Works with debugger option -g.	Does very little optimization.
1	Level 0 optimizations, plus peephole (statement-by-statement) optimizations and instruction scheduling.	Produces faster programs than level 0. Compiles faster than level 2.	Compiles slower than level 0. Does not work with debugging option -g.
2	Level 1 optimizations, plus global (whole procedure) optimizations.	Produces faster run-time code than level 1.	Compiles slower than level 1. Does not work with debugging option -g.
3	Level 2 optimizations, plus procedure integration.	Produces the fastest run-time code.	Compiles the slowest. It may increase program size and does not work with debugging option -g.

Profiling works at all optimization levels. For details on the kinds of optimizations performed at each level, see the last section in this chapter, "A Closer Look at Optimizations."

Invoking Optimization

By default, the C compiler (cc) performs level 0 optimization. To get level 1 optimization, compile the program with the +O1 option; to get level 2, compile with +O2 or synonymously, -O; to get level 3, compile with +O3.

3

For example to get level 2 optimization, use one of the following C compiler command lines (they are equivalent):

```
cc +O2 prog.c
cc -O prog.c
```

In addition to the previously mentioned options, the C compiler recognizes the +OV option, which declares all global variables as **volatile**. (For details on when to use +OV, see "Troubleshooting Optimization Problems" later in this chapter.) For instance, if you were to compile the following program with +OV:

```
int i;
float a;
main()
{
int idx;
  ⋮
}
```

It would produce the same results as applying the **volatile** qualifier to the global variables i and a:

```
volatile int i;
volatile float a;
main()
{
int idx;
  ⋮
}
```

3

Table 3-2 summarizes the optimization compiler options.

Table 3-2. Optimization Compiler Options

Option	What It Does
+O1	Invokes level 1 optimization.
+O2	Invokes level 2 optimization.
-O	Invokes level 2 optimization.
+O3	Invokes level 3 optimization.
+OV	Applies the `volatile` qualifier to all global variables.

Using Directives to Control Optimization

The C compiler provides an initial optimization level to programs through the use of special cc options (i.e., +O1, +O2, -O, +O3). To allow the user to refine the initial optimization level, the C compiler provides three directives, also known as #pragmas: OPTIMIZE, OPT_LEVEL, and NO_SIDE_EFFECTS. They must appear outside any function, and they apply for the remainder of the file or until superseded by another directive. There are also directives to control procedure inlining. These directives are HP_INLINE pragmas. These pragmas do not have to be outside of functions and only work at level +O3 optimization.

For directives to work, the source program must be compiled with one of the optimization levels listed in the previous table "Optimization Compiler Options." Otherwise, directives are ignored and warnings are issued. The remainder of this section discusses each directive in detail.

OPTIMIZE

The OPTIMIZE directive turns on or off level 2 and 3 optimization. (It does not shut off level 1 optimization.) It is useful for shutting off optimization in sections of a source program that may cause the optimizer difficulties.

Syntax

```
#pragma OPTIMIZE { ON  }
                 { OFF }
```

You should specify either ON or OFF; if you do not, the compiler generates a warning and assumes an ON setting.

Examples

#pragma OPTIMIZE ON	*restores original optimization level*
#pragma OPTIMIZE OFF	*sets optimization level to 1*

OPT_LEVEL

The OPT_LEVEL directive directs the compiler to select either level 1, 2, or 3 optimization. It is useful for switching from one level to another within a source program.

Note that it is not possible to use this pragma to raise the optimization level beyond the original level. It is only possible to lower the level or remain the same as the original level. A warning is issued if there is an attempt to raise the original optimization level.

Syntax

```
                      { 1 }
#pragma OPT_LEVEL     { 2 }
                      { 3 }
```

You should specify an optimization level; if you do not, the compiler generates a warning and assumes level 1.

Examples

```
#pragma OPT_LEVEL 1     goes to level 1 optimization
#pragma OPT_LEVEL 2     goes to level 2 optimization
#pragma OPT_LEVEL 3     goes to level 3 optimization
```

HP_INLINE_LINES

The HP_INLINE_LINES directive is used to change the maximum number of expressions a function may have and still be inlined. The default value is 20. Note that this directive only affects source code that has been compiled with level +O3 optimization.

Syntax

```
#pragma HP_INLINE_LINES n
```

The value *n* is the number of expressions that a function may have and still be inlined. Functions with more than *n* expressions will not be inlined. This value may also be set using the compiler options: `-Wi,-l`*n*

Example

`#pragma HP_INLINE_LINES 5`	*inline all functions with 5 or fewer expressions*

HP_INLINE_FORCE

The `HP_INLINE_FORCE` directive forces succeeding calls to the functions listed to be inlined regardless of the size of the function. This directive is in force from the line it first occurs on to the end of the file or until overridden by a later `HP_INLINE_FORCE`, `HP_INLINE_OMIT`, or `HP_INLINE_DEFAULT` directive. Note that this directive only affects source code that has been compiled with level `+O3` optimization.

Syntax

`#pragma HP_INLINE_FORCE` *name* [`,`*name*] ...

All calls to the function(s) specified in the *name* list will be inlined regardless of their size. Forcing functions to be in lined can also be done using this compiler option: `-Wi,-f`*name*`[,`*name*`]` ...

Example

`#pragma HP_INLINE_FORCE newfunction`	*calls to "newfunction" are inlined*
`#pragma HP_INLINE_FORCE start, function`	*calls to "start" and "function" are inlined*

HP_INLINE_OMIT

The HP_INLINE_OMIT directive forces succeeding calls to the functions listed *not* to be inlined regardless of the size of the function. This directive is in force from its point of inception until the end of the file or until overridden by a later HP_INLINE_FORCE, HP_INLINE_OMIT, or HP_INLINE_DEFAULT directive. Note that this directive only affects source code that has been compiled with level +O3 optimization.

Syntax

#pragma HP_INLINE_OMIT *name* [,*name*] ...

All calls to the function(s) specified in the *name* list will *not* be inlined. Forcing functions to *not* be inlined can also be done using this compiler option: -Wi,-o*name*,[*name*] ...

Example

`#pragma HP_INLINE_OMIT func1`	*calls to "func1" are not inlined*
`#pragma HP_INLINE_OMIT newfunc, oldfunc`	*calls to "newfunc" and "oldfunc" are not inlined*

HP_INLINE_DEFAULT

The HP_INLINE_DEFAULT directive cancels any HP_INLINE_FORCE or HP_INLINE_OMIT directives for the functions listed. Note that this directive only affects source code that has been compiled with level +O3 optimization.

3

Syntax

```
#pragma HP_INLINE_DEFAULT name [,name] ...
```

All previous HP_INLINE_FORCE and HP_INLINE_OMIT directives for the function(s) specified in the *name* list will be canceled.

Example

`#pragma HP_INLINE_DEFAULT f1, f2`	*cancels any previous force and omit directives for functions "f1" and "f2"*

3

HP_INLINE_NOCODE

The HP_INLINE_NOCODE causes stand-alone code for the functions listed to *not* be emitted. This reduces the size of the object file. The HP_INLINE_NOCODE directive *must not* be used on a function that may be called from a routine in a separate file or on a function that is not always inlined. Note that this directive only affects source code that has been compiled with level +O3 optimization.

Syntax

`#pragma HP_INLINE_NOCODE` *name* [,*name*] ...

All calls to the functions specified in the *name* list will not emit stand-alone code. Preventing stand-alone code from being emitted by functions can be done by using this compiler option: `-Wi,-n`*name*`[,`*name*`]` ...

Example

`#pragma HP_INLINE_NOCODE ft1`	*no stand-alone code is emitted for "f1"*
`#pragma HP_INLINE_NOCODE ft1, ft2`	*no stand-alone code is emitted for "f1" or "f2"*

NO_SIDE_EFFECTS

By default, the optimizer assumes that all functions *might* modify global variables. To some degree, this assumption limits the extent of optimizations it can perform on global variables. The NO_SIDE_EFFECTS directive provides a way to override this assumption. If you know for certain that some functions do *not* modify global variables, you can gain further optimization of code containing calls to these functions by specifying these functions' names to this directive.

Syntax

```
#pragma NO_SIDE_EFFECTS name [ ,name] ...
```

All functions in *name* list are the names of functions which do *not* modify the values of global variables. Global variable references can be optimized to a greater extent in the presence of calls to the listed functions. Note that you need the `NO_SIDE_EFFECTS` directive in the files where the calls are made, not where the function is defined. This directive takes effect from the line it first occurs on to the end of the file.

3

Examples

Figure 3-1 shows an example program with both correct and incorrect usage of the `NO_SIDE_EFFECTS` directive. Line 26 illustrates correct usage because it properly lists functions that do *not* modify global variables. Line 33, on the other hand, is incorrect because it lists functions that *do* modify global variables.

It is also important to note that the calls to func1 and func2 on lines 21 and 22 are not affected by the NO_SIDE_EFFECTS directive on line 26. But the calls to func2 and func1 on lines 44 and 45 are affected (that is, additional optimizations may be possible) because they follow the NO_SIDE_EFFECTS directive.

```
 1 /* global variables */
 2
 3 int glob1, glob2, glob3
 4
 5 main()
 6 {
 7 /* functions which do NOT modify globals */
 8
 9 int func1(), func2();
10
11 /* function which DO modify globals */
12
13 int func3(), func4();
14 int x,y,z;
15   :
16
17 /* These calls to func1 and func2 are NOT affected
18    by the NO_SIDE_EFFECTS directive because they
19    come before the directive. */
20
21 x = func1(0);
22 y = func2(x);
23
24 }
25
26 #pragma NO_SIDE_EFFECTS func1, func2
27
28 /* The following use of NO_SIDE_EFFECTS
29    is INCORRECT because it lists functions
30    which DO modify global variables.
31    Take care not to misuse it this way. */
32
```

```
33 #pragma NO_SIDE_EFFECTS func3, func4
34
35 int f()
36 {
37 int x,y,z;
38   :
39
40 /* But these calls ARE affected because
41    they appear after the NO_SIDE_EFFECTS
42    directive which lists them. */
43
44 z = func2(y);
45 x = func1(z);
46 }
```

Figure 3-1. Example of Using `NO_SIDE_EFFECTS`

Error Messages

Levels 2 and 3 Optimization Errors

During levels 2 and 3 optimization, the compiler may produce any of several messages. Listed below are the possible messages and a description of each:

- `Global optimizer warning in command line:`
  `-Wg,-d has been specified. CSE disabled.`

  Common subexpression elimination is disabled. If the `-Wg` and `-d` command-line options or the `CCOPTS` environment variable is removed, common subexpression elimination will be reenabled.

- `Global optimizer warning in "`*procname*`":`
  `Possible infinite loop detected.`

  The global optimizer has detected an infinite loop. Note that the optimizer assumes that all function calls, including calls to routines such as `exit(2)`, will return control to the calling routine.

- `Global optimizer warning in "`*procname*`":`
  `Loop increment of 0 detected.`

  A loop increment of 0 was found. This may cause an infinite loop.

- `Global optimizer warning in "`*procname*`":`
  `Variable "`*var name*`" uninitialized before use.`

  The named variable is referenced without a prior assignment in any of the control paths leading to the reference.

  The absence of this message does not ensure that all variables are properly initialized. In particular, assignments on some, but not all control paths leading to a reference will cause this message not to be printed.

  Formal parameters and global variables are implicitly assigned at the beginning of a function, and will never appear in this message.

- `Global optimizer warning in "`*procname*`":`
  `Location "`*offset*`(%a6)" uninitialized before use.`

  An uninitialized stack location was referenced, but no variable name could be matched to it.

- `Global optimizer warning in "`*procname*`": Procedure not reducible (no global optimizations or register allocation performed in this routine).`

  The control flow through the routine was so irregular that the optimizer could not discern normal loop patterns. This is usually caused by poorly structured code with `goto`s. Only common subexpression elimination was performed within each basic block. No other optimizations or register allocation was done for this routine. For more information on this topic, read the section "Non-Reducible Code" found in this chapter.

- `Global optimizer error: out of space - unable to allocate memory for internal use.`

  This message indicates that the optimizer was unable to acquire the necessary amount of dynamic memory space. For more information on this, read the section "What to Do About Slow Compilation and Out-of-Memory Conditions" found in this chapter.

- `Global optimizer warning in "`*procname*`":`
  `Division by 0`

  The global optimizer has detected an apparent division by 0. This condition may have been uncovered after optimizations such as constant folding or constant propagation in which a division by a variable is reduced to division by a constant 0.

- `Global optimizer warning in "`*procname*`":`
  `Exceeded default complexity level for loop optimization.`
  `Use -Wg,-All to override.`

  The global optimizer encountered a function whose complexity is such that full loop optimization may take an excessive amount of time to complete. Therefore, it will not attempt the most time consuming loop optimizations such as code motion and strength reduction on this function. Other optimizations and register allocations are still performed. In many cases of this nature, the optimizations will have little effect on the overall performance of a program. Note that the default limits may be overridden, however, a longer compilation time should be expected.

  This warning occurs only when the verbose flag (`-v`) is used on the `cc` command line.

- `C1 internal error in "`*procname*`": ...`
  `C1 internal error in command line: ...`

  These messages indicate an abnormal condition was detected within the optimizer. These messages also indicate a defect in the optimizer that should be reported to your HP support representative. A workaround is to not use optimization or select level 1 optimization.

Errors Caused by Pre-6.5 Compilers

The linker will complain about any attempt to use a pre-6.5 module by issuing the following warning:

`ld: (warning) old (pre-6.5) file` *file_name* `may be incompatible with newer files`

What to Do About Slow Compilation and Out-of-Memory Conditions

In order to perform transformations that involve entire functions, the global optimizer must maintain large internal data structures. Compiling large functions, especially those containing many loops or branches, may occasionally take excessive time to compile or result in a "`Global optimizer error: out of space - unable to allocate memory for internal use`" error. In either case, several remedies are available:

- If possible, split the large function into smaller pieces. Compilation speed will improve greatly and the smaller routines are frequently easier to understand and maintain.
- Use lower-level optimization on the appropriate file or function. You can:
  - Compile with the `-O` option instead of the option `+O3`
  - Remove the `-O` option (for level 0) or change it to `+O1` (for level 1) on the command line to select the optimization level for the entire file, or
  - Leave the `-O` option and surround the large function with a `#pragma OPT_LEVEL 1`, `#pragma OPT_LEVEL 2` pair; for example:

```
    ⋮
/* Very large function follows; set to level 1. */

#pragma OPT_LEVEL 1

int  very_larg_func(arg1, arg2, arg3, arg4)
int  arg1, arg2;
char *arg3;
char arg4;
{
   ⋮  /* body of the very large function */
}

/* Out of the function, so reset to level 2. */

#pragma OPT_LEVEL 2
    ⋮
```

- Compile with the `+OV` option to avoid optimizing references to global variables. Often this significantly reduces the size of optimizer internal data structures.

The "`out of space`" message means the optimizer was unable to acquire the necessary amount of dynamic memory space. This may be caused by insufficient physical memory or disk swap space. The amount of data memory space allowed per process may be increased by altering the kernel configuration parameter, `maxdsiz` (refer to the *HP-UX System Administrator Task Manual*). Swap space can be increased by configuring additional swap space disks or by reformatting the disk with more swap space.

In some cases, the system administrator can enlarge the swap area without reconfiguring the file system. For more information on how to enlarge the swap area, read about the *swapon (1M)* command in the *HP-UX Reference*.

Troubleshooting Optimization Problems

Occasionally a program works differently after optimization. If this happens do the following:

3

1. Run the `lint` program (see *lint(1)* in the *HP-UX Reference*) to catch common programming errors. Pay particular attention to function calls with too few arguments. The called function will use unrelated contents of the stack for the remaining arguments, and the stack contents are likely to be different when the program is optimized. Behavior may also be different if a programming error causes an out-of-bounds array access. All references of the form `*(p+i)` are assumed to remain within the bounds of the variable to which `p` points.
2. Make sure that all variables are correctly initialized. With level 2 optimization (because of extensive use of registers), an uninitialized variable is usually assigned whatever is left in the register; this value is usually non-zero. The level 2 optimizer will print a warning if it can determine that no path within the program will initialize a variable before use; it cannot detect uninitialized global variables or local variables that are initialized in only certain sequences of program execution.
3. Verify that the problem is not simply a floating point precision change due to heavier use of MC6888*x* floating point registers. Level 0 and 1 optimized floating point code usually has many more MC6888*x* register loads and stores, with accompanying 80-bit to 64/32-bit conversions. A mathematically ill-conditioned solution may be adversely affected by the added precision of fewer loads and stores.
4. Make sure that optimizer assumptions are not violated. Signal handlers that modify global variables and memory-mapped input/output may require the use of the `volatile` qualifier on the declaration of affected variables. If many global variables are modified by signal handlers, you may also want to consider compiling with the `+OV` option.
5. Try optimizing at level 1. If possible, isolate the problem to a single routine and optimize all others at level 2 or above.
6. If none of these solutions works, try level 0 optimization.

A Closer Look at Optimizations

This section describes the code transformations performed during optimization:

- peephole optimization
- instruction scheduling.
- constant folding
- constant propagation
- dead code elimination
- coloring register allocation
- common subexpression elimination
- copy propagation
- loop unrolling
- code motion of loop invariants
- strength reduction of loop induction variables
- tail recursion elimination
- non-reducible code
- procedure integration

Note An optimizer doesn't actually change the C source code; it performs transformations internally.

Peephole Optimization

During compilation, the C compiler produces assembly language code. Peephole optimization looks at a small window of this assembly language code, looking for optimization opportunities. Wherever possible, the peephole optimizer replaces assembly language instruction sequences with faster (usually shorter) sequences, and removes redundant register loads and stores.

For example, the code

```
fpmov.s  L14,%fpa0
fpadd.s  %fpa0,%fpa1
```

is changed to

```
fpmadd.s L14,%fpa0,%fpa1
```

Peephole optimization is performed at levels 1 and above.

The peephole optimizer C2 also performs instruction scheduling. The scheduler takes advantage of the processors ability to concurrently execute floating point and integer instructions, and attempts to avoid sequences that cause A-register or FP-register interlocks which result in a pipline stall.

The C2 optimizer is invoked for all levels of optimization (i.e., +O1, +O2, +O3, and -O), and by default always performs instruction scheduling. This feature may be disabled by the -i option to C2 (or W2,-i option to cc).

Instruction Scheduling

Instruction scheduling is the reordering of the instructions within a basic block (a straight line piece of code which is entered only at the top and exits only at the bottom) to increase execution speed while producing the same result. The implementation details of the processor's instruction pipeline can result in two different orderings of the same instruction sequence executing at different speeds. For the MC68030 and MC68882 combination, performance gains can be achieved by scheduling integer instructions to execute concurrently with floating point instructions and by separating floating point instructions in which the destination register of the first instruction is the source register of the next instruction. For the MC68040, additional speed up can be achieved by separating instructions in which the destination of the first instruction is an A-register and the source address of the second instruction includes indirection through that same A-register. For example, the following C program:

```
int i,j,k;
float a,b,c;

main()
{
   i = j + k;
   a = b + c;
}
```

produces assembly instructions for the two assignment statements as follows:

```
mov.l    _j,%d0
add.l    _k,%d0
mov.l    %d0,_i
fmov.s   _b,%fp0
fadd.s   _c,%fp0
fmov.s   %fp0,_a
```

The instruction scheduler reorders the instructions as follows:

```
fmov.s    _b,%fp0
fadd.s    _c,%fp0
mov.l     _j,%d0
add.l     _k,%d0
mov.l     %d0,_i
fmov.s    %fp0,_a
```

thus, allowing the evaluation of the statement `i = j + k` to execute concurrently with the evaluation of the expression `b + c`.

Constant Folding

Constant folding computes the value of a constant expression at compile time. For example,

```
secs_per_hr = 60*60;
```

is replaced by

```
secs_per_hr = 3600;
```

Constant folding is performed at all optimization levels, but more opportunities may arise in combination with other level 2 optimizations.

Constant Propagation

Performed at levels 2 and above only, constant propagation replaces a variable with a constant value, if that can be determined at compile time. For example, after constant propagation and constant folding, the code

```
a = 1;
b = 2;
c = a + b;
```

becomes

```
a = 1;
b = 2;
c = 3;
```

Dead Code Elimination

Performed at levels 2 and above only, dead code elimination removes code that cannot be reached. For example, the code

3

```
#define BUFFER_SIZE 512
int a[BUFFER_SIZE];

deadcode()
{
int request = 256;

 if (BUFFER_SIZE >= request)
  initialize(a);
 else
  expand_and_init(a);
}
```

after constant propagation and dead code elimination becomes

```
int a[512];
deadcode()
{
 initialize(a);
}
```

Coloring Register Allocation

Performed at levels 2 and above only, coloring register allocation replaces memory references to variables and constants with references to hardware registers. It does this only for those variables for which it would be most advantageous. The entire function is examined to determine the best variables for register allocation. The name of this optimization comes from the similarity to map coloring algorithms in graph theory.

Simple register allocation is done at levels 0 and 1. Integer variables (including `char`, `short`, and `long`) are assigned to available MC680x0 "D" registers when the storage class specifier `register` is included in the variable's declaration. Similarly, pointer variables are assigned to MC680x0 "A" registers. No floating point variables are assigned to registers at levels 0 and 1.

At levels 2 and above optimization, candidates for register assignment are selected based on estimated speed improvement rather than by the `register` declaration. Preference is given to variables that are used frequently or that appear within a loop. Floating point variables can be assigned to registers. Structure fields and array elements are not eligible for register assignment. The benefits of full register allocation over levels 0 and 1 are that it eliminates the need to request the `register` storage class in the source program, and it is able to reuse registers within a block. Furthermore, frequently used global variables can also be allocated a register. For example, in the following code, the same register can be used for both loop indexes; at level 0 or 1, a register declaration would have to be included, and two registers would have been reserved for `i` and `j` for the duration of the function.

```
int a[5], b[5];
reuse()
{
int i,j;
 for (i=0; i<5; i++) a[i]=0;
 for (j=0; j<5; j++) b[j]=0;
}
```

Common Subexpression Elimination

Performed at levels 2 and above only, common subexpression elimination locates repeated operations and uses the saved result rather than duplicating the operation. These types of operations include loading a value from memory into a register, as well as arithmetic computations.

For example, the code

```
a = x + y + z;
b = x + y + w;
```

becomes

```
t1 = x + y;
a = t1 + z;
b = t1 + w;
```

Dead Store Elimination

Performed at levels 2 and above only, dead store elimination removes assignments to variables that are not used again in the program unit. For example, the following function

3

```
f(x)
int x;
{
int a;
 a = 1;
 return (x);
}
```

becomes

```
f(x)
int x;
{
 return (x);
}
```

Copy Propagation

If a variable is assigned an expression and used once later in a section of straight-line code, copy propagation replaces the use of the variable with the expression that was assigned to it. Copy propagation may permit subsequent constant folding and dead store elimination. For example, the code below

```
int a,b;
copy_propagation()
{
 int t;
 t = 3 * a;
 b = 4 * t;
}
```

becomes

```
int a,b;
copy_propagation()
{
 b = 12 * a;
}
```

Copy propagation is performed at levels 2 and above only.

Loop Unrolling

Performed at levels 2 and above only, loop unrolling expands a short loop into an equivalent sequence of instructions. This may have a cost of increased code size, but it saves incrementing and testing loop indexes and creates the possibility for other optimizations.

For example,

```
int a[3];
unroll()
{
int i;
 for (i=0; i<3; i++)  a[i] = 2*i;
}
```

becomes

```
int a[3];
unroll()
{
 a[0] = 0;
 a[1] = 2;
 a[2] = 4;
}
```

Code Motion of Loop Invariants

Performed at levels 2 and above only, code motion moves statements or statement fragments that are relatively constant in a loop to the outside of the loop, thus ensuring that they are executed only once. For example,

```
code_motion(x, a, b, c)
int x[100];
int a, b, c;
{
  int i;
  int tot;

  tot = 0;
  for (i =1; i < 100; i++)
    {
      x[i] = a*b+c;
      tot += x[i];
    }
  return(tot);
}
```

this function can be changed to read as shown below after some dead storage elimination and replacing the relatively constant right-hand side of the assignment to x[i]:

```
code_motion(a, b, c)
int a, b, c;
{
  int i;
  int tot;
  int temp;

  tot = 0;
  temp = a*b+c;
  for (i =1; i < 100; i++)
    {
      tot += temp;
    }
  return(tot);
}
```

Strength Reduction of Loop Induction Variables

Performed at levels 2 and above only, strength reduction replaces time consuming arithmetic operations with less expensive ones. For example in the `strength_reduction` function given below, a multiply which is inherent in the array reference is replaced by a much quicker add.

```
strength_reduction()
{
  int x[100][100];
  int i, j;

  for (i = 0; 1 < 100; i++)
    for (j = 0; j < 100; j++)
      x[i][j] = i + j;
  return(x[99][99]);
}
```

The above function is internally transformed into the following function after strength reduction.

```
strength_reduction()
{
  int x[100, 100];
  int i, j;
  int **ptr;

  for (i = 0; ptr = x; 1 < 100; i++)
    for (j = 0; j < 100; j++)
      **ptr++ = i + j;
  return(x[99][99]);
}
```

Elimination of Tail Recursion

Performed at levels 2 and above only, the optimizer detects certain forms of recursive programs and rewrites them to use local branching instead, thus reducing the call overhead. For example,


```
frecursive(c, f)
int c;
void (*f)();

{
  (*f)(c);
  c +=1;
  frecursive(c, f);
}
```

becomes, by eliminating the recursive call to `frecursive()`,

```
frecursive(c, f)
int c;
void (*f)();

  {
    top:
         (*f)(c);
         c += 1;
         goto top;
  }
```

which is much faster because no intermediate calls are required.

Non-Reducible Code

Performed at level 2 and above only, the optimizer will not optimize procedures that contain a non-reducible flow graph. Non-reducible flow graphs are blocks of code that form a loop which can be entered from more than one point. For example, this short program contains a non-reducible flow graph:

```
 1:   int i;
 2:   main ()
 3:   {
 4:      if ( i == 99 )
 5:         goto label;
 6:      while (1)
 7:         {
 8:            if ( i == 10 )
 9:               i = 99;
10:            else
11:   label:          i = 999;
12:         }
13:   }
```

Note that in this program the `if ... else` statement of lines 8 through 11 can be entered from the `while` statement on line 6 or from the `goto` statement on line 5. A flow graph of this loop would look like this:

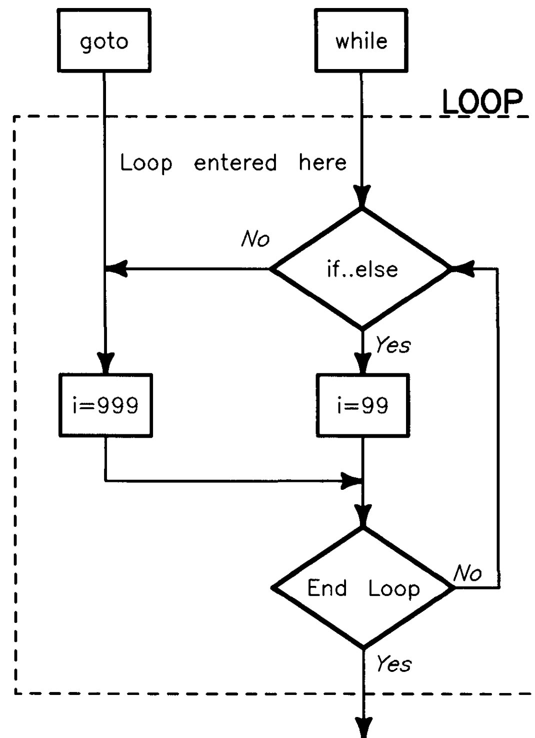


Figure 3-2. Non-Reducible Flow Graph

If you compile the previously given program using the following command:

```
cc -O flow.c
```

you will get this message:

```
Global optimizer warning in "main": Procedure not reducible
(no global optimizations or register allocation performed in this routine).
```

To allow the procedure to be optimized, use this command:

```
cc -O -Wg,-x flow.c
```

3

Procedure Integration

This section deals with procedure integration replacing function calls with actual code from the called function. This allows for faster execution by removing the overhead of a function call and may also allow increased global optimization.

Integration of System Functions

When compiling with the optimization option +O3, certain system function calls are replaced by the actual code in order to speed up execution. The system calls that may be replaced with inline code are:

strcpy	acos	asin	atan	cos
cosh	exp	fabs	log	log10
sin	sinh	sqrt	tan	tanh

Note The math functions (all but strcpy) are replaced by inlined code that does not exactly duplicate the behavior of the library code. They do not set errno or call matherr in the case of error conditions. Some of them may generate a floating point exception when called with an illegal parameter (e.g., taking the sqrt of a negative value).

Integration of User Defined Functions

User defined functions can also be inlined when compiling with the optimization option +O3. To replace a call to a user defined function with the actual function code requires that both the call and the text of the function being called be in the same file. Also all parameters must match with respect to number and type. A list of directives to control which functions are and are not inlined is given in the section "Using Directives to Control Optimization" found in this chapter.

Inlining user defined functions can increase the code size of a program as multiple calls to a function are replaced by copies of the called function. For this reason inlining should only be done on small functions or where a function is called many times during program execution.

4

Implementation Dependencies

This chapter describes implementation dependencies for C 7.0 and 8.0.

Implementation Dependencies for C 7.0

Primary Name Definitions in C Libraries

In order to comply with the ANSI-C and POSIX 1003.1 standards, the `libc.a` and `libm.a` libraries will now use primary and secondary definitions in referring to library routines. The primary definition is the name of a library routine prefixed with one or more underscores; the secondary definition is the name.

Your programs may be affected if you have replaced a library routine by defining your own function with the same name. Beginning with 7.0 if you do not use the primary definition name, instead of linking to your replacement, your program will link to the library version. If the primary definition is used in your replacement, your program will work just as it did before 7.0, but it will be in violation of ANSI-C standards.

The primary and secondary definitions of a routine can be found by using `nm(1)` on the library. The secondary definition will have an *S* next to the segment; the primary definition will not.

The two exceptions to the new primary definitions are the `matherr(3M)` and `malloc(3C)` routines. Because so many users write their own versions, they will still be referenced in the old way. Programs that define their own `matherr()`, `malloc()`, `free()`, and/or `realloc()` will not have to change for 7.0.

Implementation Dependencies for C 8.0

Support for Shared Libraries

Shared libraries is a new HP-UX 8.0 feature. The Series 300/400 and Series 600/700/800 C languages provide the following support for this feature:

- Position independent code (PIC) generation
- A `pragma` that associates a shared library version with a module.

Position Independent Code

Series 300/400 and Series 600/700/800 C compilers provide `+z` and `+Z` as options to generate position independent code.

For more details on shared libraries and these options, read the *Programming on HP-UX* manual and the *HP-UX Reference* page for the *cc(1)* command.

Version `pragma`

Series 300/400 and Series 600/700/800 C compilers provide a new `pragma` called `HP_SHLIB_VERSION`. This `pragma` associates a shared library version number with a module. The syntax of the pragma is:

```
#pragma HP_SHLIB_VERSION "date"
```

where the format of *date* is `mm/yy` (month/year). The version number is derived from the *date* and has file scope which means that it applies to all exported objects in the source file.

5

Porting to ANSI C

This chapter describes the process of moving existing programs to ANSI C. In particular, this chapter discusses:

- The `const` and `volatile` qualifiers
- How to upgrade existing programs to use function prototypes
- How name spaces work for ANSI C and other standards
- Cases where ANSI C behavior is silently different from existing C.

For information on implementation-defined behaviors and extensions for the Series 300/400 ANSI-C product, read Appendix A in this manual.

The `const` and `volatile` Qualifiers

C on Series 300/400 computers includes two new keywords from the ANSI C definition: `const` and `volatile`. Used in variable declarations, these keywords qualify (or modify) the way in which the compiler treats the declared variable; as such, they are known as *qualifiers*. They are reserved keywords and may not be used as variable or function names.

The const Qualifier

The `const` qualifier declares *constant variables*—that is, variables whose values cannot be changed during program execution. Attempting to assign a value to a `const` variable causes a compile error. For instance, the following statement declares a constant variable `pi` of type `float` with initial value 3.14:

```
const float pi = 3.14;
```

A constant variable can be used like any other constant; for example:

```
area = pi * (radius * radius);
```

But attempting to assign a value to a `const` variable causes a compile error:

```
pi = 3.1416;     /* this causes an error */
```

5

However, the compiler detects only obvious attempts to modify `const` variables. For example, given the declaration

```
float *ptr;
```

the following code alters the contents of `pi` without error:

```
ptr = &pi;
*ptr = 2.7;
```

Note that `const` can also be used on pointer types to declare constant pointers; for example:

```
char *const prompt = "Hello> ";
```

Any obvious attempt to reassign the pointer `prompt` will cause a compile error; for instance:

```
prompt = "Good-bye> ";  /* This will cause an error. */
```

The volatile Qualifier

With the 6.5 release and later, the C compiler includes a more powerful optimizer. To do its best job, the optimizer makes assumptions about how variables are used within a program. For example, it assumes that the contents of memory will not be changed by entities other than the current program. Signal handlers that alter global variables or memory-mapped input/output may violate this assumption.

The `volatile` qualifier provides a way to "mark" variables that may violate optimizer assumptions. When the optimizer encounters a `volatile` variable it does not make its normal assumptions and, thus, is more conservative when optimizing statements that reference that variable. For example, the following code fragment marks an `int` array named `foo` as being volatile:

```
volatile int foo[100];
```

Note that the `volatile` qualifier can also be applied to pointer types; for example:

```
volatile *char version_info = "Release 1.01 - 010188";
```

For details on using the optimizer, its assumptions, and when to use the `volatile` qualifier, refer to the chapter "Optimizing C Programs."

Upgrading Existing C Programs to Use Prototypes

For the most part, existing programs will compile unchanged in ANSI C. However, ANSI C has introduced a new syntax for declaring functions. The new syntax for a function declaration defines a function and its parameters and their types. This function declaration is called a *function prototype.*

Advantages of the Function Prototype

Adding function prototypes to existing C programs can yield three advantages:

- Better type checking between declarations and calls because the number and types of the parameters are part of the function's parameter list. For example,

5

```
struct s
  {
   int i;
  };

int old_way(x)
  struct s x;
  {
   /* Function body using the old method for
      declaring function parameter types
   */
  }

int new_way(struct s x)
  {
   /* Function body using the new method for
      declaring function parameter types
   */
  }

/* The functions "old_way" and "new_way" are
   both called later on in the program.
*/
```

```
old_way(1);  /* This call compiles without complaint. */
new_way(1);  /* This call gives an error. */
```

In this example, the function `new_way` gives an error because the value being passed to it is of type `int` instead of type `struct x`.

- More efficient parameter passing in some cases. Parameters of type `float` are not converted to `double`. For example,

```
void old_way(f)
  float f;
  {
   /* Function body using the old method for
      declaring function parameter types
   */
  }

void new_way(float f)
  {
   /* Function body using the new method for
      declaring function parameter types
   */
  }

/* The functions "old_way" and "new_way" are
   both called later on in the program.
*/

float g;

old_way(g);
new_way(g);
```

In the above example, the function `old_way` is called, `g` is converted to a `double` and pushed on the stack. The `old_way` function then pops `g` off the stack and converts it to `float`. When the function `new_way` is called, `g` gets pushed on the stack without any conversion and `new_way` then pops `g` off the stack. This function is more efficient because the data type of `g` remains unchanged.

- Automatic conversion of function arguments, as if by assignment. For example, integer parameters may be automatically converted to floating point.

```
/* Function declaration using the new method
   for declaring function parameter types
*/

extern double sqrt(double);

/* The function "sqrt" is called later
   on in the program.
*/

sqrt(1);
```


  In this example, any value passed to `sqrt` is automatically converted to `double`.

Compiling an existing program in ANSI mode will yield some of these advantages because of the existence of prototypes in the standard header files. To take full advantage of prototypes in existing programs, change old style declarations (i.e. without prototype) to new style declarations. The tool `protogen` (see *protogen(1)* in the online man pages) helps add prototypes to existing programs. For each source file, `protogen` can produce a header file of prototypes and a modified source file that includes prototype declarations.

Function Prototype Considerations

There are three things to consider when using function prototypes.

- Type Difference between Actual and Formal Parameters
- Declaration of a Structure in a Prototype Parameter
- Mixing of `const` and `volatile` Qualifiers and Function Prototypes

Type Difference between Actual and Formal Parameters

When a prototype to a function is added, one should be careful that all calls to that function occur with the prototype visible (in the same context). The following example illustrates problems that may arise when this is not the case:

```
func1(){
  float f;
  func2(f);
 }

int func2(float arg1){
   /* body of func2 */
}
```

In the example above, when the call to func2 occurs, the compiler behaves as if func2 had been declared with an old-style declaration int func2(). For an old-style call, the default argument conversion rules cause the parameter f to be converted to double. When the declaration of func2 is seen, there is a conflict. The prototype indicates that the parameter arg1 should not be converted to double, but the call in the absence of the prototype indicates that arg1 should be widened. When this conflict occurs within a single file, the compiler will issue an error:

```
function prototype for 'func2' should contain parameters compatible
with default argument promotions when used with an empty declaration.
```

This error may be fixed by either making the prototype visible before the call, or by changing the formal parameter declaration of arg1 to double. If the declaration and call of func2 were in separate files, then the compiler would not detect the mismatch and the program would silently behave incorrectly.

The *lint(1)* command can be used to find such parameter inconsistencies across files.

Declaration of a Structure in a Prototype Parameter

Another potential prototype problem occurs when structures are declared within a prototype parameter list. The following example illustrates a problem that may arise:

```
func3(struct stname *arg);
struct stname { int i; };

void func4(void) {
   struct stname s;
   func3(&s);
}
```

In this example, the call and declaration of `func3` are not compatible because they refer to different structures, both named `stname`. The `stname` referred to by the declaration was created within prototype scope. This means it goes out of scope at the end of the declaration of `func3`. The declaration of `stname` on the line following `func3` is a new instance of `struct stname`. When conflicting structures are detected the compiler will issue an error:

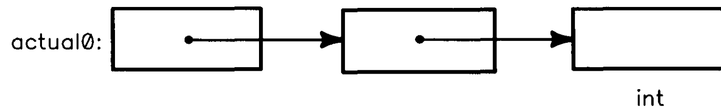
```
types in call and definition of 'func3' have incompatible
struct/union pointer types for parameter 'arg'
```

This error may be fixed by switching the first two lines and thus declaring `struct stname` prior to referencing it in the declaration of `func3`.

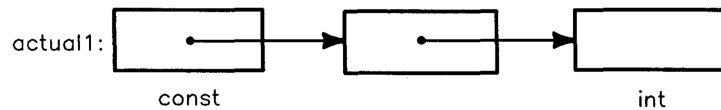
Mixing of const and volatile Qualifiers and Function Prototypes

Mixing the `const` and `volatile` qualifiers and prototypes can be tricky. Note that this section uses the `const` qualifier for all of its examples; however, you could just as easily substitute the `volatile` qualifier for `const`. The rules for prototype parameter passing are the same as the rules for assignments. To illustrate this point, consider the following declarations:

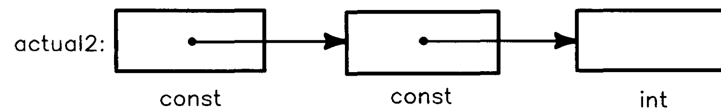
```
/* pointer to pointer to int */
int **actual0;
```



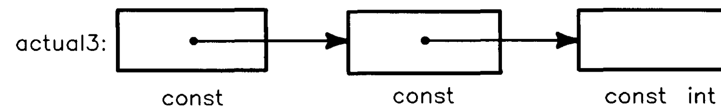

```
/* const pointer to pointer to int */
int **const actual1;
```




```
/* const pointer to const pointer to int */
int *const *const actual2;
```




```
/* const pointer to const pointer to const int */
const int *const *const actual3;
```



These declarations show how successive levels of a type may be qualified. The declaration for `actual0` has no qualifiers. The declaration of `actual1` has only the top level qualified. The declarations of `actual2` and `actual3` have two and three levels qualified. When these actual parameters are substituted into calls to the following functions:

```
void f0(int **formal0);
void f1(int **const formal1);
void f2(int *const *const formal2);
void f3(const int *const *const formal3);
```

The compatibility rules for pointer qualifiers are different for all three levels. At the first level, the qualifiers on pointers are ignored. At the second level, the qualifiers of the formal parameter must be a superset of those in the actual parameter. At levels three or greater the parameters must match exactly. Substituting `actual0` through `actual3` into `f0` through `f3` results in the following compatibility matrix:

	f0	**f1**	**f2**	**f3**
actual0	C	C	C	N
actual1	C	C	C	N
actual2	S	S	C	N
actual3	NS	NS	N	C

C = compatible

S = not compatible, qualifier level two of formal is not a superset of actual parameter

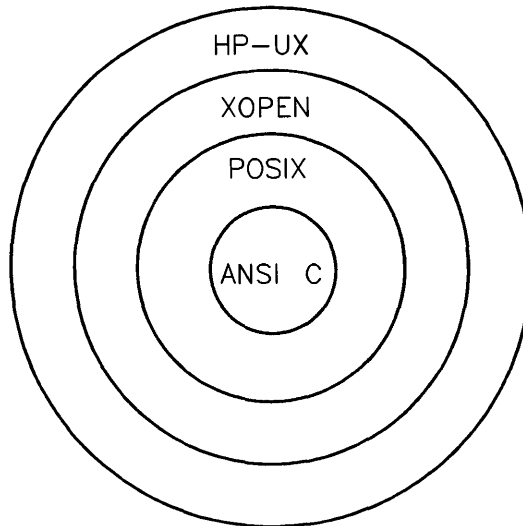
N = not compatible, qualifier level three doesn't match

How the Name Spaces Work for ANSI C and Other Standards

The ANSI C standard specifies exactly which names are reserved by the implementation (compiler, libraries and header files). These reserved names are given a special *name space* by the ANSI C implementation. The intention is to make it easier to port programs from one implementation to another with no fear of unexpected collisions in names. For example, since the ANSI C standard does not reserve the keyword `open`, an ANSI C program may define and use a function named `open` without colliding with the *open(2)* system call in different operating systems.

HP Header File and Library Implementation of Name Space

The HP header files and libraries have been designed to support four different name spaces.



where:

ANSI C — is the set of names defined in the ANSI C standard.

POSIX — is the set of names defined in the POSIX 1003.1 standard. These names are a superset of those used by ANSI C.

XOPEN — is the set of names defined by the XOPEN standard. These names are a superset of those used by POSIX.

HP-UX — is all names defined in the header files, a superset of XOPEN.

The HP library implementation has been designed with the assumption that many existing programs will use more routines than those allowed by the ANSI C standard. If a program calls, but does not define a routine that is not in the ANSI C name space (e.g. `open`), then the library will resolve that reference. This allows a clean name space and backward compatibility.

The HP header file implementation uses a set of predefined names to select the name space. In compatibility mode the default is the HP-UX name space. Compatibility mode means that virtually all programs that compiled and executed under previous releases of the HP C Language on HP-UX will continue to work as expected. The following table provides information on how to select a name space from a command line or from within a program using the defined libraries.

Table 5-1. Selecting a Name Space in ANSI Mode

When using the name space ...	Use command line option ...	or #define in source program
HP-UX	`-D _HPUX_SOURCE`	`#define _HPUX_SOURCE`
XOPEN	`-D _XOPEN_SOURCE`	`#define _XOPEN_SOURCE`
POSIX	`-D _POSIX_SOURCE`	`#define _POSIX_SOURCE`
ANSI C	default	default

In ANSI mode, the default is ANSI C name space. The symbols _POSIX_SOURCE, _XOPEN_SOURCE or _HPUX_SOURCE may be used to select other name spaces. The _HPUX_SOURCE symbol may need to be defined to make existing programs compile in ANSI mode. For example,

```
#include <sys/types.h>
#include <sys/socket.h>
```

will result in the following compile-time error in the ANSI mode because `socket.h` uses the symbol `u_short` and `u_short` is only defined in the HP-UX name space section of `types.h`:

```
"/usr/include/sys/socket.h", line 79: syntax error:
   u_short sa_family;
         ^
```

5

This error may be fixed by adding `-D _HPUX_SOURCE` to the command line of the compile.

Silent Changes for ANSI C

This section describes the situations that occur when the ANSI mode silently has different behavior from the compatibility mode. Many of these silent behaviors can be detected by running the *lint(1)* program. The following list provides some of these silent behaviors:

- A bit field declared without the `signed` or `unsigned` keywords will be `signed` in ANSI mode and `unsigned` in the compatibility mode.
- Trigraphs are new in ANSI C. A trigraph is a three character sequence that is replaced by a corresponding single character. For example, ??= is replaced by #. For more information on trigraphs, read *C: A Reference Manual.*

- Promotion rules for `unsigned char` and `unsigned short` have changed. Compatibility mode rules specify when an `unsigned char` or `unsigned short` is used with an integer the result is `unsigned`. ANSI-mode rules specify the result is `signed`. The following program example illustrates a case where these rules are different.

```
main(){
  unsigned short us = 1;
  int i = -2;
  printf("%s\n",(i+us)>0 ? "compatibility mode" : "ansi mode");
}
```

  Note that differences in promotion rules may occur under the following conditions:[1]

  - An expression involving an `unsigned char` or `unsigned short` produces an integer-wide result in which the sign bit is set: that is, either a unary operation on such a type, or a binary operation in which the other operand is `int` or "narrower" type.
  - The result of the preceding expression is used in a context in which its condition of being signed is significant: it is the left operand of the right-shift operator or either operand of /,%,<,<=,>, or >=.

[1] *Rationale for Draft Proposed American National Standard for Information Systems - Programming Language C* (311 First Street, N.W., Suite 500, Washington, DC 20001-2178; X3 Secretariat: Computer and Business Equipment Manufactures Association), pages 34 - 35.

- Floating point expressions with `float` parameters may be computed as `float` precision in ANSI mode. In compatibility mode they will always be computed in `double` precision.
- Initialization rules are different in some cases when braces are omitted in an initialization.
- Unsuffixed integer constants may have different types. In compatibility mode, unsuffixed constants have type `int`. In the ANSI mode, unsuffixed constants less than or equal to 2147483647 have type `int`. Constants larger than 2147483647 have type `unsigned`. For example:

  ```
  -2147483648
  ```

  has type `unsigned` in the ANSI mode and `int` in compatibility mode. The above constant is `unsigned` in the ANSI mode because `2147483648` is `unsigned`, and the `-` is a unary operator.

- Empty tag declarations in a *block scope* create a new `struct` instance in the ANSI mode. The term *block scope* refers to identifiers declared inside a block or list of parameter declarations in a function definition that have meaning from their point of declaration to the end of the block. In the ANSI mode, it is possible to create recursive structures within an inner block. For example,

  ```
  struct x { int i; };
  { /* inner scope */
    struct x;
    struct y { struct x *xptr; };
    struct x { struct y *yptr; };
  }
  ```

  In the ANSI mode, the inner `struct x;` declaration creates a new version of the structure which may then be referred to by `struct y`. In compatibility mode, the `struct x;` declaration refers to the outer structure and the program is incorrect. For more information, read the section "Structure Type Reference" in the chapter "Types" in *C: A Reference Manual.*

A

Implementation-Defined Behaviors and Extensions to ANSI-C

This appendix contains information on all implementation-defined behaviors and extensions for the Series 300/400 ANSI-C product. Except where noted these characteristics hold true for both compatibility and ANSI mode.

Implementation-Defined Behaviors

Diagnostic Messages

Description	Behavior

Format of Diagnostic Messages	Diagnostic messages may be either errors or warnings.
	Error messages have this format:
	`"filename", line number : message`
	Warning messages have this format:
	`"filename", line number : warning : message`

Arguments to main ()

Description	Behavior
Semantics of arguments to `main()`	`argc`, `argv`, and `envp` are passed, and are writable. The arguments from the command line invocation are stored in `argv` (one argument per entry). The number of elements is contained in `argc`. The strings are case sensitive.

Interactive Device

Description	Behavior
Whether data will be line buffered or block buffered	■ If *isatty(3C)* returns true, then data is line buffered. ■ If *isatty(3C)* returns false, then data is block buffered.

A

Identifiers

Description	Behavior
Number of significant initial characters in an identifier without external linkage	255
Number of significant initial characters in an identifier with external linkage	255
Significance of case distinction in identifiers with external linkage	The case is significant for external identifiers

Handling Characters

Description	Behavior
Source and execution character sets: the source character set is the set of characters in which source files are written and the execution character set is the set of characters that may be used during execution of a program.	The source and execution character sets are ASCII. The members of the source set depend on the language used. For NLS processing, any character that is valid in that language may be used in strings, character constants, and header file names. The source character set does not include: `\x01`, `\x02`, `\x03`, and `\xff`. If these characters are seen in the input character set, `cpp` issues a diagnostic message and ignores the characters. The execution character set includes all other 8-bit values. Source characters are mapped one-for-one into the execution character set.
Value of an integer character constant that contains more than one character wide character constant or that contains more than one multibyte character	A warning is given for character constants that contain more than one character.
Current locale used to convert multiple characters into corresponding wide characters for a wide character constant	Uses the LC_CTYPE environment variable at compile time

Handling Characters (continued)

Description	Behavior
Whether a "plain" `char` has the same range of values as `signed char` or `unsigned char`	`char` has the same set of values as a `signed char`

Handling Integers

Description	Behavior
Representations of the various types of integers	2's complement representation is used char - 8 bits short - 16 bits int - 32 bits long - 32 bits
Result of converting an integer to a shorter signed integer	Preserves the value of the sign bit and the least significant bits
Result of converting an unsigned integer to its corresponding signed integer of equal length	The most significant bit of the unsigned value becomes the sign bit
Results of bitwise operations on signed integers	Uses 2's complement representation
Sign of the remainder on integer division	The sign of the remainder is the same as the sign of the dividend.
Result of a right shift of a negative-valued signed integral type	This is an arithmetic shift which shifts in the sign bit.

Handling Floating-Point Values

Description	Behavior
Representations of the various floating-point numbers	Series 300/400 computer representations follow the IEEE 754 standard; however, the representations are not fully implemented for NAN, denormalized numbers, and infinities.
Direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value	Rounds toward the nearest value. If two values are equidistant from the value being rounded, then the one with a zero (0) in the least significant bit of the mantissa is chosen.
Direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number	The truncation is the same as for integral numbers.

Handling Arrays and Pointers

Description	Behavior
Type of integer required to hold the maximum size of an array	Unsigned integer
Result of casting a pointer to an integer or vice versa	The bits from pointers copy directly to *int* and vice versa.
Type of integer required to hold the difference between two pointers to elements of the same array	Uses the type *int*

Registers

A

Description	Behavior
Extent to which objects can actually be placed in registers by use of the register storage-class specifier.	There are 6 data and 4 address (or 3 with `+ffpa`) registers available. ■ Only scalar types may be placed in registers (i.e. no `struct`, `union`, or floating point). ■ Pointers are placed in address registers until they are used up. ■ Integral types are placed in data registers until they are used up. ■ With optimization at level 2 or 3, register declarations are ignored and the optimizer performs register allocations.

Handling of Structures, Unions, Enumerations and Bit Fields

Description	Behavior
Padding and alignment of members of structures	See the chapter in this manual "Data Type Alignments"
Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field	The `int` bit-field is treated as signed
Order of allocation of bit-fields within an `int`	The order of allocation is from left to right (most significant to least significant).
Crossing of bit fields over storage-unit boundaries	Bit fields cannot cross boundaries.
Integer type chosen to represent the values of an enumeration type	Uses *int*.

A

Qualifiers

Description	Behavior
Access to an object that has a `volatile` qualified type	Reads or writes of all of or part of that object; however, `volatile` objects are not placed in registers

Declarator Limits

Description	Behavior
Maximum number of declarators (i.e., array of, "[ ]"; pointer to, "*"; function returns, "( )") that may modify an arithmetic, structure, or union type.	The maximum number is 13.

Case Limits

Description	Behavior
Maximum number of case values in a *switch* statement.	Has no arbitrary limit

A

Preprocessing Directives

Description	Behavior
Determine if the value of a single-character constant in a constant expression (that controls conditional inclusion) matches the value of the same character constant in the execution character set	Has one-to-one mapping from source character set to execution character set; therefore, their values match
Determine if the value of a single-character constant in a constant expression (that controls conditional inclusion) can have a negative value.	It can have a negative value for a single-character constant in a constant expression.
Method for locating includable source files	See the *cpp(1)* command in the *HP-UX Reference*
Support of the `#pragmas` directives	The following `#pragmas` are supported: HP_ALIGN HP_INLINE_DEFAULT HP_INLINE_FORCE HP_INLINE_LINES HP_INLINE_NOCODE HP_INLINE_OMIT HP_SHLIB_VERSION NO_SIDE_EFFECTS OPTIMIZE [ON|OFF] OPT_LEVEL [0|1|2|3]
Definitions for _DATE_ and _TIME_ when the date and time of translation are not available	The null string ("").

A

Library Functions

Description	Behavior
Null-pointer constant to which the macro NULL expands	0
Diagnostic printed by, and the termination behavior of the assert function (also note that the *abort(3C)* function is invoked after the assertion failure)	Assertion failed:<text of expression>, file <file name>, line <line number>
Sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions	The set of characters depend on the locale used. The default values for C are: isalnum: '0'-'9', 'A'-'Z', 'a'-'z' isalpha: 'A'-'Z', 'a'-'z' iscntrl: 0x0 - 0x2F islower: 'a'-'z' isupper: 'A'-'Z' isprint: ' '-'~'
Values returned by the mathematical functions on domain errors	NAN when libM.a is used Zero (0) when libm.a is used
Whether mathematic functions set errno on underflow range errors	errno is assigned the value of the macro ERANGE on underflow range errors.
Effect the fmod function has when its second argument is 0	Zero (0) is returned
Set of signals for the signal function	See *signal(5)* in the *HP-UX Reference*

Description	Behavior
Semantics for each signal recognized by the `signal` function	See *signal(5)* in the *HP-UX Reference*
Default handling and the handling at program start-up for each signal recognized by the `signal` function	See *signal(5)* in the *HP-UX Reference*
Whether the equivalent of `sig(`*sig*`,SIG_DFL)` is executed prior to the call of a signal handler; the blocking of the signal that is performed	See *signal(5)* in the *HP-UX Reference*
Effects of default handling if the SIGILL signal is received by the handler specified to the *signal* function	See *signal(5)* in the *HP-UX Reference*
Need for a new-line to terminate the last line of a text stream file	A new-line is not required
Space characters that are written out to a text stream immediately before a new-line character is read in	Space characters appear
Number of null characters that may be appended to data written to a binary stream	No null characters are appended
Location of the file position indicator of an append mode stream	It is initially positioned at the end of the file
Effect of doing a write on an open text stream when the current file pointer is not at the end of the file	■ If the mode was append (a or a+) then writes are always appended to the end of the file regardless of the value of the file pointer ■ If the mode was write (w or w+) or r+, the write is done to the location of the current file pointer. The file is not truncated beyond that point.

Library Functions (continued)

Description	Behavior
Buffering of file	Supports unbuffered, buffered and line buffered characteristic
Existence of zero-length files	Supports zero-length files
File name rules	Supports HP-UX file naming rules (see the section "Naming Files" in the chapter "Working with Files" found in *A Beginner's Guide to HP-UX*
Opening a file multiple times	Supports opening a file multiple times
Effects of the `remove` function on an open file	This function removes an open file.
Effects of executing the `rename` function with a file name that exists	The function will be executed, and it will overwrite the file that already exists assuming the directory has write permission.
Output for %p conversion in the `fprintf` function	A sequence of eight unsigned hexadecimal digits is output. If the value being converted can be represented in fewer digits, it will be expanded with leading zeros.
Input for %p conversion in the `fscanf` function	A sequence of up to eight unsigned hexadecimal digits is read.

A

Description	Behavior
Interpretation of a character that is neither the first nor the last character in the scan list for `%[]` conversion in the `fscanf` function	The construct [*first* - *last*] represents a range of characters where *first* must be lexically less than or equal to *last*; The lexical order is determined by the program's locale (LC_COLLATE category). For example, the scan set `[a-z]` will be all lower case if the LC_COLLATE category is set to the C locale. It will not be all lower case if the LC_COLLATE category is set to the `french@fold` locale.
Value to which the `fgetpos` or `ftell` function set the ERRNO macro when a failure occurs	Same possible values as those set by *lseek(2)*: EBADF - fildes is not an open file descriptor ESPIPE - fildes is associated with a pipe or FIFO EINVAL - is not one of the supported value of the resulting file offset would be negative
Messages generated by the `perror` function	See *perror(3)* in the *HP-UX Reference* or the file `/usr/include/sys/errno.h`

A

Library Functions (continued)

Description	Behavior
Behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero	For `malloc`, size=0 causes 0 bytes to be memory allocated (may be grown with `realloc`). For `realloc`, size=0 causes the object to be freed. For `calloc`, size=0 causes the object to be memory allocated.
Behavior of the `abort` function with regard to open and temporary files	This function closes all files.
Status returned by the `exit` function if the value of the argument is other than zero, EXIT_SUCCESS or EXIT_FAILURE	`exit` returns no value (it cannot return to its caller), see the *HP-UX Reference* for details.
Altering the environment name with the `getenv` function	The environment name is whatever is passed in by the `exec` command (set up in shell environments, etc). The method of modifying the environment is to use *putenv(3C)* call.
Contents and mode of execution of the string by the `system` function	It is given to `sh` as input.
Messages generated by the `strerror` function	See *strerror(3)* in the *HP-UX Reference* or the file `/usr/include/sys/errno.h`

A

Description	Behavior
Local time zone and Daylight Savings Time	Set according to the TZ environment variable
Era for the clock function	See *time(2)* in the *HP-UX Reference*

Implementation-Defined Extensions

HP supports the following extensions to ANSI C.[1] Extensions that are compatible with the ANSI C standard are on by default. Other extensions are turned on with the `+e` option. Use of these extensions may reduce the portability of programs.

Implementation-Defined Extensions

Extension	Behavior
Environment Arguments	The main function receives a third argument `char *envp[]` that points to a null-terminated array of strings that provide information about the environment for the execution of the process.
Scopes of Identifiers	An identifier which contains the keyword `extern` has file scope.

A

[1] The material in this section comes from the *American National Standard for Information Systems - Programming Language C, ANS X3.159-1989* (311 First Street, N.W., Suite 500, Washington, DC 20001-2178; X3 Secretariat: Computer and Business Equipment Manufactures Association).

Implementation-Defined Extensions (continued)

Extension	Behavior
Writable string literals	String literals are modifiable. Identical strings are distinct.
Function pointer casts	Pointers may be cast from objects to functions and vice-versa. This allows data to be invoked as a function, and functions to be examined as objects.
Non-`int` bit fields	Any integral type may be declared as a bit field.
`asm`	Text between the delimiters `asm("` and `")` is inserted directly in the assembly file. This option is incompatible with ANSI C.
Common Storage Model	There may be more than one external definition for an object with or without the keyword `extern`. The definitions may not disagree and no more than one may be initialized.
Predefined Macro Names	See *cpp(1)* in the *HP-UX Reference*.
`$` as an identifier character	This extension accepts the `$` character as a valid identifier in a character string as long as it is not the first character (e.g., `a$b` is valid).

A

B

HP-UX Reference Pages

This appendix provides reference pages for C and ANSI C commands.

NAME

cc, c89 – C compiler

SYNOPSIS

cc [*options*] *files*
c89 [*options*] *files*

DESCRIPTION

cc is the HP-UX C compiler. *c89* is the HP-UX POSIX conforming C compiler. Both accept several types of arguments as *files*:

- Arguments whose names end with **.c** are understood to be C source files. Each is compiled and the resulting object file is left in a file having the corresponding basename, but suffixed with **.o** instead of **.c**. However, if a single C file is compiled and linked, all in one step, the **.o** file is deleted.
- Similarly, arguments whose names end with **.s** are understood to be assembly source files and are assembled, producing a **.o** file for each **.s** file.
- Arguments whose names end with **.i** are assumed to be the output of *cpp*(1) (see the **–P** option below). They are compiled without again invoking *cpp*(1). Each object file is left in a file having the corresponding basename, but suffixed **.o** instead of **.i**.
- Arguments of the form **–l***x* cause the linker to search the library **lib***x***.sl** or **lib***x***.a** in an attempt to resolve currently unresolved external references. Because a library is searched when its name is encountered, placement of a **–l** is significant. If a file contains an unresolved external reference, the library containing the definition must be placed *after* the file on the command line. See *ld*(1) for further details.
- All other arguments, such as those whose names end with **.o** or **.a**, are taken to be relocatable object files that are to be included in the link operation.

Arguments and options can be passed to the compiler through the **CCOPTS** environment variable as well as on the command line. The compiler reads the value of **CCOPTS** and divides these options into two sets; those options which appear before a vertical bar (|), and those options which appear after the vertical bar. The first set of options are placed before any of the command-line parameters to *cc*; the second set of options are placed after the command-line parameters to *cc*. If the vertical bar is not present, all options are placed before the command-line parameters. For example (in *sh*(1) notation),

```
CCOPTS="–v |–lmalloc"
export CCOPTS
cc –g prog.c
```

is equivalent to

```
cc –v –g prog.c –lmalloc
```

When set, the TMPDIR environment variable specifies a directory to be used by the compiler for temporary files, overriding the default directories **/tmp** and **/usr/tmp**.

Options

It should be noted that from the *cc* and *c89* options; **–A, –G, –g, –O, –p, –v, –y, +z, +Z** are not supported by the C compiler provided as part of the standard HP-UX operating system. They are supported by the C compiler sold as an optional separate product.

The following option is recognized only by *cc*:

–A*mode* Specify the compilation standard to be used by the compiler. The *mode* can be one of the following letters:

- **c** Compile in a mode compatible with HP-UX releases prior to 7.0. (See *The C Programming Language*, First Edition by Kernighan and Ritchie). This option is currently the default. The default may change in future releases.
- **a** Compile under ANSI mode (ANSI programming language C standard ANS X3.159-1989). When compiling under ANSI mode, header files define only those names specified by the standard. To get the same name space as in compatibility mode (**–Ac**), define the symbol _HPUX_SOURCE.

The following options are recognized by both *cc* and *c89*:

−c Suppress the link edit phase of the compilation, and force an object (**.o**) file to be produced for each **.c** file even if only one program is compiled. Object files produced from C programs must be linked before being executed.

−C Prevent the preprocessor from stripping C-style comments (see *cpp*(1) for details).

−D*name*=*def*
−D*name* Define *name* to the preprocessor, as if by '#define'. See *cpp*(1) for details.

−E Run only *cpp*(1) on the named C or assembly files, and send the result to the standard output.

−g Cause the compiler to generate additional information needed by the symbolic debugger. This option is incompatible with optimization.

−G Prepare object files for profiling with *gprof* (see *gprof*(1)).

−I*dir* Change the algorithm used by the preprocessor for finding include files to also search in directory *dir*. See *cpp*(1) for details.

−l*x* Refer to item (4) at the beginning of the DESCRIPTION section.

−L *dir* Change the algorithm used by the linker to search for **lib***x***.sl** or **lib***x***.a**. The **−L** option causes *cc* to search in *dir* before searching in the default locations. See *ld*(1) for details.

−n Cause the output file from the linker to be marked as shareable. For details and system defaults, see *ld*(1).

−N Cause the output file from the linker to be marked as unshareable. For details and system defaults, see *ld*(1).

−o*outfile* Name the output file from the linker *outfile*. The default name is **a.out**.

−O Invoke the optimizer with level 2 optimization. Equivalent to +O2.

−p Arrange for the compiler to produce code that counts the number of times each routine is called. Also, if link editing takes place, replace the standard startoff routine by one that automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

−P Run only *cpp*(1) on the named C files and leave the result on corresponding files suffixed **.i**. The **−P** option is also passed along to *cpp*(1).

−q Cause the output file from the linker to be marked as demand loadable. For details and system defaults, see *ld*(1).

−Q Cause the output file from the linker to be marked as not demand loadable. For details and system defaults, see *ld*(1).

−s Cause the output of the linker to be stripped of symbol table information. See *strip*(1) for more details. The use of this option prevents the use of a symbolic debugger on the resulting program. See *ld*(1) for more details.

−S Compile the named C files, and leave the assembly language output on corresponding files suffixed **.s**.

−tx,name Substitute subprocess *x* with *name* where *x* is one or more of a set of identifiers indicating the subprocess(es). This option works in two modes: 1) if *x* is a single identifier, *name* represents the full path name of the new subprocess; 2) if *x* is a set of identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path names of the new subprocesses.

The *x* can take one or more of the values:

p Preprocessor (standard suffix is **cpp**)
c Compiler (standard suffix is **ccom**)

0 Same as **c**
a Assembler (standard suffix is **as**)
l Linker (standard suffix is **ld**)

−**U***name* Remove any initial definition of *name* in the preprocessor. See *cpp*(1) for details.

−v Enable verbose mode, which produces a step-by-step description of the compilation process on the standard error.

−w Suppress warning messages.

−W *x,arg1*[,*arg2*...]
Pass the argument[s] *argi* to subprocess *x*, where *x* can assume one of the values listed under the **−t** option as well as **d** (driver program). The **−W** option specification allows additional, implementation-specific options to be recognized by the compiler driver. For example,

−Wl,−a,archive

causes the linker to link with archive libraries instead of with shared libraries. See *ld*(1) for details. For some options, a shorthand notation for this mechanism can be used by placing "+" in front of the option name as in

+M

which is equivalent to

−Wc,−M

+M is the Series 300/400 option which causes the compiler to generate calls to the math library instead of generating code for the MC68881 or MC68882 math coprocessor. Options that can be abbreviated using "+" are implementation dependent, and are listed under DEPENDENCIES.

−y Generate additional information needed by static analysis tools, and ensure that the program is linked as required for static analysis. This option is incompatible with optimization.

−Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See *hpnls*(5) for more details on International Support.

−z Do not bind anything to address zero. This option allows runtime detection of null pointers. See the note on *pointers* below.

−Z Allow dereferencing of null pointers. See the note on *pointers* below. The **−z** and **−Z** are linker options. See *ld*(1) for more details.

+z,+Z Both of these options cause the compiler to generate position independent code (PIC) for use in building shared libraries. The options **−g** , **−G** , **−p** , and **−y** are ignored if **+z** or **+Z** are used. Normally, **+z** should be used to generate PIC; however, when certain limits are exceeded, **+Z** is required to generate PIC. The linker *ld*(1) issues the error indicating when **+Z** is required. If both **+z** and **+Z** are specified, only the last one encountered applies. For a more complete discussion regarding PIC and these options, see the manual *Programming on HP-UX*.

Any other options encountered generate a warning to standard error.

Other arguments are taken to be C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name **a.out**.

The first edition of "The C Programming Language", by Kernighan and Ritchie, and the various addenda to it, are intentionally ambiguous in some areas. HP-UX specifies some of these below for compatibility mode (**−Ac**) compilations.

char The **char** type is treated as signed by default. It may be declared **unsigned**.

pointers Accessing the object of a NULL (zero) pointer is technically illegal (see Kernighan and Ritchie), but many systems have permitted it in the past. The following is provided to maximize portability of code. If the hardware is able to return zero for reads of

location zero (when accessing at least 8- and 16-bit quantities), it must do so unless the **−z** flag is present. The **−z** flag requests that SIGSEGV be generated if an access to location zero is attempted. Writes of location zero may be detected as errors even if reads are not. If the hardware cannot assure that location zero acts as if it was initialized to zero or is locked at zero, the hardware should act as if the **−z** flag is always set.

identifiers
: Identifiers are significant up to 255 characters.

types
: Certain programs require that a type be a specific number of bits wide. It can be assumed that an **int** can hold at least as much information as a **short**, and that a **long** can hold at least as much information as an **int**. Additionally, either an **int** or a **long** can hold a pointer.

EXTERNAL INFLUENCES

Environment Variables

When the **−Y** option is invoked, LC_CTYPE determines the interpretation of string literals and comments as single and/or multi-byte characters.

LANG determines the language in which messages are displayed.

If LC_CTYPE is not specified in the environment or is set to the empty string, the value of LANG is used as a default for each unspecified or empty variable. If LANG is not specified or is set to the empty string, a default of "C" (see *lang*(5)) is used instead of LANG. If any internationalization variable contains an invalid setting, *cc* behaves as if all internationalization variables are set to "C". See *environ*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasionally, messages may be produced by the preprocessor, assembler or the link editor.

If any errors occur before *cc* is completed, a non-zero value is returned. Otherwise, zero is returned.

EXAMPLES

The following compiles the C file **prog.c**, to create a **prog.o** file, and then invoke the link editor *ld*(1) to link **prog.o** and **procedure.o** with all the C startup routines in **/lib/crt0.o** and library routines from the C library **libc.sl** or **libc.a**. The resulting executable program is output in **prog**:

```
cc prog.c procedure.o −o prog
```

WARNINGS

Options not recognized by *cc* are not passed on to the link editor. The option **−W l,***arg* can be used to pass any such option to the link editor.

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to explicitly call *exit*(2) or to leave the function **main()** with a '**return** *expression*;' construct.

DEPENDENCIES

Series 300/400

It should be noted that from the following Series 300/400-specific *cc* and *c89* options; **+e, +O, +y** are not supported by the C compiler provided as part of the standard HP-UX operating system. They are supported by the C compiler sold as an optional separate product.

The **−z** option is not supported.

The default is to allow null pointer dereferencing, hence using **−Z** has no effect.

The compiler supports the following additional options. The *+opt1* notation can be used as a shorthand notation for some **−W** options.

+bfpa
: Cause the compiler to generate code that uses the HP98248A or HP98248B floating point accelerator card, if it is installed at run time. If the card is not installed, floating point operations are done on the MC68881 or MC68882 math coprocessor or the MC68040.

+e *or* **−W c,−We** Enables HP value added features when compiling in ANSI C mode, **−Aa.** This option is ignored with **−Ac** since these features are already provided. Features enabled:

- **$** as an identifier character
- Accept embedded assembly code

+ffpa Cause the compiler to generate code for the HP98248A or HP98248B floating point accelerator card. This code does not run unless the card is installed.

+M Cause the compiler not to generate inline floating point code for the MC68881 , MC68882 or MC68040. Library routines are referenced for *matherr* capability.

+N*secondary N* Adjust the initial size of internal compiler tables. *secondary* is one of the letters from the set {**abdepstw**}, and *N* is an integer value. *secondary* and *N* are *not* optional. The Series 300/400 compiler automatically expands the tables if they become full. The +N option is supported only for backwards compatibility.

+O*opt* Invoke optimizations selected by *opt*. If *opt* is **1**, only level 1 optimizations are handled. If *opt* is **2**, all optimizations except inlining are performed. The **−O** option is equivalent to **+O2**. If *opt* is **V**, optimization level 2 is selected, but all global variables and objects dereferenced by global pointers are treated as if they were declared with the keyword "volatile," meaning that references to the object cannot be optimized away. If *opt* is **3**, all level 2 optimizations are performed and in addition, code for certain functions is generated in line rather than calling the function. Functions that are 'inlined' are *strcpy,* the transcendental functions available on the MC68881 or MC68882 math coprocessor, and certain user-defined functions. For a complete discussion of the various optimization levels, see the *C Programmers Guide*.

+s By default, compilation subprocesses are run concurrently and, in ANSI mode, **cpp** and **ccom (cpass1)** are merged into a single subprocess. This results in better compile time performance except when available compilation memory is scarce. Invoking this option executes the processes sequentially and executes **cpp** and **ccom (cpass1)** as distinct processes, thereby minimizing memory consumption.

−t*x,name* Specify additional subprocess identifiers.

- **0** First pass of the compiler with level 2 optimization. It is not the same as subprocess **c** (standard suffix is **cpass1**)
- **1** Second pass of the compiler with level 2 optimization (standard suffix is **cpass2**)
- **g** Level 2 global optimizer (standard suffix is **c.c1**)
- **2** Peephole optimizer (standard suffix is **c.c2**)
- **i** Procedure integrator (standard suffix is **c.c0**)

−v Enables verbose mode in the global optimizer as well.

−W c,−F Perform some function inlining. The functions that are 'inlined' are *strcpy,* and the transcendental functions available on the MC68881 or MC68882 math coprocessor.

−W c,−YE Cause source code lines to be printed on the assembly (**.s**) file as assembly comments, thus showing the correspondence between C source and the resulting assembly code. This option is incompatible with optimization.

−W g,−All Cause the global optimizer to apply all optimizations. By default, the global optimizer does not attempt certain optimizations when the complexity of a function exceeds a certain limit. This option causes the global optimizer to unconditionally apply all optimizations.

+y The default behavior for generating symbolic debugging information (**−g**) and static analysis information (**−y**) is to generate such information only for items referenced in the file being compiled. For example, if a structure is defined in

some included header file yet never referenced, no symbolic debugging information or static analysis information is generated for that structure. The **+y** option causes the compiler to generate symbolic debugging information or static analysis information for all items, whether referenced or not. The **+y** option is only valid when used with **−g** or **−y**.

Series 700/800

It should be noted that from the following Series 700-and-800-specific *cc* and *c89* options, **+e, +O, +y** are not supported by the C compiler provided as part of the standard HP-UX operating system. They are supported by the C compiler sold as an optional separate product.

The default is to allow null pointer dereferencing, hence using **−Z** has no effect.

The **−g** option is incompatible with optimization. If both debug and optimization are specified, only the first option encountered takes effect.

The **−y** option is incompatible with optimization. If both static analysis and optimization are specified, only the first option encountered takes effect.

The **−s** option is incompatible with the **−g** , **−G** , **−p** , and **−y** options. If **−s** is specified along with any of the above options, the **−s** option is ignored, regardless of the order in which the options were specified.

Nonsharable, executable files generated with the **−N** option cannot be executed via *exec*(2). For details and system defaults, see *ld*(1).

The compiler supports the following additional options. The *+opt1* notation can be used as a shorthand notation for some **−W c** options.

−W*d,−a* — When processing files which have been written in assembly language, does not assemble with the prefix file which sets up the space and subspace structure required by the linker. Files assembled with this option cannot be linked unless they contain the equivalent information.

+DA*architecture* — Generate code for the *architecture* specified. *architecture* is required. The default code generated for the Series 800 is **PA_RISC_1.0.** The default code generated for the Series 700 is **PA_RISC_1.1.** The default code generation may be overriden using the **CCOPTS** environment variable or the command line option **+DA.** Defined values for *architecture* are:

- **1.0** Precision Architecture RISC, version 1.0.
- **1.1** Precision Architecture RISC, version 1.1.

The compiler determines the target architecture using the following precedence:

1. Command line specification of **+DA**.
2. Specification of **+DA** in the **CCOPTS** environment variable.
3. The default as mentioned above.

+DS*architecture* — Use the instruction scheduler tuned to the *architecture* specified. *architecture* is required. If this option is not used, the compiler uses the instruction scheduler for the architecture on which the program is compiled. Defined values for *architecture* are:

- **1.0** Precision Architecture RISC, version 1.0.
- **1.1** Precision Architecture RISC, version 1.1, general scheduling for the series 700.
- **1.1a** Scheduling for specific models of Precision Architecture RISC, version 1.1.

+e *or* **−W c,−e** — Enables HP value added features while compiling in ANSI C mode, **−Aa.** This option is ignored with **−Ac** since these features are already provided. Features enabled:

- Long pointers
- Missing parameters on intrinsic calls

+L *or* **−W c,−L** Enable the listing facility and any listing pragmas. A straight listing prints:

- A header on the top of each page
- Line numbers
- The nesting level of each statement
- The postprocessed source file with expanded macros, included files, and no user comments (unless the −C option is used).

If the **−Aa** option is used to compile under ANSI C, the listing shows the original source file rather than the postprocessed source file.

+Lp Print a listing as described above, but show the postprocessed source file even if one of the ANSI compilation levels is selected. This option is ineffective if the **−y** option is used.

+m *or* **−W c,−m** Cause the identifier maps to be printed. First, all global identifiers are listed, then all the other identifiers are listed by function at the end of the listing. For struct and union members, the address column contains B@b, where B is the byte offset and b is the bit offset. Both B and b are in hexadecimal.

+o *or* **−W c,−o** Cause the code offsets to be printed in hexadecimal; they are grouped by function at the end of the listing.

+O*opt*
or
−W c,−O*opt* Invoke optimizations selected by *opt*. If *opt* is **1**, only level 1 optimizations are handled. If *opt* is **2**, all optimizations are performed. The option **+O2** is the same as **−O**. If *opt* is **V** all memory references are treated as if they were declared with the keyword "volatile," meaning that references to the object cannot be optimized away.

+Obb*num* Specify the maximum number of basic blocks allowed in a procedure which is to be optimized at level 2. A basic block is a sequence of code with a single entry point and single exit point, with no internal branches. Optimizing procedures with a large number of basic blocks can take a long time and may use a large amount of memory. If the limit is exceeded, a warning is emitted giving the name of the procedure and the number of basic blocks it contains; level 1 optimization is performed. The default value for this limit is 500. This option implies level 2 optimization (equivalent to **−O** or **+O2**).

+r *or* **−W c,−r** Inhibits the automatic promotion of float to double when evaluating expressions and passing arguments. This option is ignored and a warning produced if the **−Aa** option is in effect.

+R*num* *or* **−W c,−R***num*
Allow only the first num 'register' variables to actually have the 'register' class. Use this option when the register allocator issues an "out of general registers" message.

+u Allow pointers to access members of non-natively aligned structs and unions.

+w*n* *or* **−W c,−wn**
Specify the level of the warning messages. The value of *n* can be one of the following values:

- **1** All warnings are issued.
- **2** Only warnings indicating that code generation might be affected are issued. Equivalent to the compiler default without any **w** opts.
- **3** No warnings are issued. Equivalent to the **−w** option.

+y Generate static analysis information for all global identifiers not seen in the original source file. This option only has effect if used in conjunction with the **−y** option.

FILES

file.c	input file
file.o	object file

a.out	linked output
/tmp/ctm*	default temporary files
/usr/tmp/ctm*	default temporary files
/lib/ccom	C compiler
/lib/cpp	preprocessor
/lib/cpp.ansi	preprocessor for ANSI C
/bin/as	assembler, *as*(1)
/bin/ld	link editor, *ld*(1)
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling via *prof*(1)
/lib/gcrt0.o	startoff for profiling via *gprof*(1)
/lib/libc.a	standard C library (archive version), see *HP-UX Reference* Section (3).
/lib/libc.sl	standard C library (shared version), see *HP-UX Reference* Section (3).
/lib/libp/libc.a	C library for profiled programs (archive version)
/usr/include	standard directory for **#include** files
Series 300/400	
/lib/ccom.ansi	ANSI C compiler
/lib/cpass1	pass 1 of the optimizing compiler
/lib/cpass1.ansi	pass 1 of the optimizing ANSI compiler
/lib/cpass2	pass 2 of the optimizing compiler
/lib/c.c0	procedure inliner
/lib/c.c1	global optimizer
/lib/c.c2	peephole optimizer
Series 700/800	
/usr/lib/nls/$LANG/cc.cat	C Compiler message catalog

SEE ALSO

adb(1), as(1), cdb(1), cpp(1), gprof(1), ld(1), prof(1), exit(2), crt0(3), end(3C), monitor(3C), matherr(3M).

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

American National Standard for Information Systems – Programming language C, ANS X3.159-1989

STANDARDS CONFORMANCE

cc: SVID2, XPG2, XPG3
c89: POSIX

NAME

cxref − generate C program cross-reference

SYNOPSIS

cxref [*options*] *files*

DESCRIPTION

cxref analyzes a collection of C files and attempts to build a cross-reference table. *cxref* utilizes a special version of *cpp* to include **#defined** information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or with the **−c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference. Output is sorted in ascending collation order (see Environment Variables below).

In addition to the **−D**, **−I** and **−U** options (which are identical to their interpretation by *cc*(1)), the following *options* are interpreted by *cxref*:

−c Print a combined cross-reference of all input files.

−w*num* Width option; format output no wider than *num* (decimal) columns. This option defaults to 80 if *num* is not specified or is less than 51.

−o *file* Direct output to the named *file*.

−s Operate silently; do not print input file names.

−t Format listing for 80-column width.

−Aa Choose ANSI mode. If not specified, compatibility mode (**−Ac** option) is selected by default.

−Ac Choose compatibility mode. This option is selected by default if neither **−Aa** nor **−Ac** is specified.

EXTERNAL INFLUENCES

Environment Variables

LC_COLLATE determines the order in which the output is sorted.

If LC_COLLATE is not specified in the environment or is set to the empty string, the value of LANG is used as a default. If LANG is not specified or is set to the empty string, a default of "C" (see *lang*(5)) is used instead of LANG. If any internationalization variable contains an invalid setting, *cxref* behaves as if all internationalization variables are set to "C" (see *environ*(5)).

International Code Set Support

Single- and multi-byte character code sets are supported with the exception that multi-byte character file names are not supported.

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files, anyway.

EXAMPLES

Create a combined cross-reference of the files **orange.c**, **blue.c**, and **color.h**:

```
cxref -c orange.c blue.c color.h
```

Create a combined cross-reference of the files **orange.c**, **blue.c**, and **color.h**: and direct the output to the file **rainbow.x**:

```
cxref -c -o rainbow.x orange.c blue.c color.h
```

WARNINGS

cxref considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#include*s **ctype.h** will contain many declarations of the variable **c**.

cxref uses a special version of the C compiler front end. This means that a file that will not compile probably cannot be successfully processed by *cxref*. In addition, *cxref* generates references *only* for those source lines that are actually compiled. This means that lines that are excluded by **#ifdef**s and the like (see *cpp*(1)) will not be cross-referenced.

cxref does not parse the CCOPTS environment variable.

FILES

/lib/cpp C-preprocessor.

/lib/xpass Compatibility-mode special version of C compiler front end.

/lib/xpass.ansi
ANSI-mode special version of C compiler front end.

SEE ALSO

cc(1), cpp(1).

STANDARDS CONFORMANCE

cxref: SVID2, XPG2, XPG3

NAME

lint − a C program checker/verifier

SYNOPSIS

lint [*options*] *file* ...

DESCRIPTION

lint attempts to detect features in C program *files* that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Program anomalies currently detected include unreachable statements, loops not entered at the top, automatic variables declared but not used, and logical expressions whose value is constant. Usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are assumed to be C source files. Arguments whose names end with **.ln** are assumed to be the result of an earlier invocation of *lint* with either the **−c** or the **−o** option used. **.ln** files are analogous to **.o** (object) files produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

lint takes all the **.c**, **.ln**, and **llib-l***x***.ln** files (specified by **−l***x* and processes them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **−p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **−c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **−c** option is used, all **.ln** and **llib-l***x***.ln** files are ignored.

Any number of *lint* options can be used, in any order, intermixed with file name arguments. The following options are used to suppress certain kinds of complaints:

−a Suppress complaints about assignments of long values to variables that are not long.

−b Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* often result in many such complaints).

−h Do not apply heuristic tests that attempt to intuitively find bugs, improve style, and reduce waste.

−u Suppress complaints about functions and external variables used and not defined or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)

−v Suppress complaints about unused arguments in functions.

−x Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

−l*x* Include additional lint library **llib-l***x***.ln**. For example, to include a lint version of the Math Library **llib-lm.ln**, insert **−lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multiple-file projects.

−n Do not check compatibility against either the standard or the portable lint library.

−p Attempt to check portability to other dialects of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

−s Make stricter checks about pointer and structure alignments that can prevent portability. Complain about a cast that converts a pointer from a less restrictive alignment to a more restrictive alignment. Complain about a structure member whose offset is not a multiple of its size.

−c Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

−o lib Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **−c** option nullifies any use of the **−o** option. The resulting lint library serves as input to *lint*'s second pass.

The **−o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use the **−x** option. The **−v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available by using "lint comments" (see below).

−A*mode* Specify the compilation standard to be used by *lint.* The *mode* can be one of the following letters:

c Process in a mode compatible with HP-UX releases prior to 7.0. (See *The C Programming Language*, First Edition by Kernighan and Ritchie). This option is currently the default. The default may change in future releases.

a Process under ANSI mode (December 7, 1988 Draft proposed ANSI C standard.)

−Y Enable support of 16-bit characters inside string literals and comments. Note that 8-bit parsing is always supported. See *hpnls*(5) for more details on international language support.

The **−D**, **−U**, and **−I** options of *cpp*(1) and the **−g**, and **−O**, options of *cc*(1) are also recognized as separate arguments. The **−g** and **−O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbols **_lint** and **_LINT_** are defined to allow certain questionable code to be altered or removed for *lint*. In addition, the pre-processor symbol **lint** is defined in compatibility mode. By default, the lint library **llib-lc.ln** encodes the HP-UX namespace version of **libc.a** . Other standards can be checked by including the appropriate **−D** option on the *lint*(1) command line. For example,

```
lint -D_POSIX_SOURCE file.c
```

reprocesses **llib−lc** to reflect the POSIX standard.

Certain conventional comments in the C source change the behavior of *lint*:

/*NOTREACHED*/ at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions such as *exit*(2)).

/*VARARGS*n**/ suppresses the usual checking for variable numbers of arguments in the following function definition. This comment must be placed just before the actual code for a function. It is not used before **extern** declarations of the same function elsewhere. The data types of the first *n* arguments are checked; a missing *n* is assumed to be 0.

/*ARGSUSED*/ enables the **−v** option for the next function.

/*LINTLIBRARY*/ at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **−v** and **−x** options.

lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **−c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name is printed, followed by a question mark.

Behavior of the **−c** and **−o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **−c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **−c** option), listing all the **.ln** files with the needed **−l***x* options. This prints all the inter-file inconsistencies. This scheme works well with *make*(1); allowing *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were processed by *lint*.

EXTERNAL INFLUENCES

Environment Variables

LC_CTYPE determines the interpretation of comments and string literals as single- and/or multi-byte characters.

If LC_CTYPE is not specified in the environment or is set to the empty string, the value of LANG is used as a default for each unspecified or empty variable. If LANG is not specified or is set to the empty string, a default of "C" (see *lang*(5)) is used instead of LANG. If any internationalization variable contains an invalid setting, *lint* behaves as if all internationalization variables are set to "C". See *environ*(5).

When set, the TMPDIR environment variable specifies a directory to be used for temporary files, overriding the default directories **/tmp** and **/usr/tmp**.

Long error messages are split across lines to make them easier to read. The environment variable COLUMNS controls the maximum number of characters on each line.

International Code Set Support

Single- and multi-byte character code sets are supported within file names, comments, and string literals.

FILES

/usr/lib	the directory where the lint libraries specified by the **−l***x* option must exist
/usr/lib/lint[12]	first and second passes
/usr/bin/lint	shell script that invokes lint[12]
/usr/lib/llib−lc.ln	declarations for C Library functions (binary format; source is in **/usr/lib/llib−lc**)
/usr/lib/llib−port.ln	declarations for portable functions (binary format; source is in **/usr/lib/llib−port**)
/usr/lib/llib-lm.ln	declarations for Math Library functions (binary format; source is in **/usr/lib/llib-lm**)
/usr/tmp/*lint*	temporaries

WARNINGS

exit(2), *longjmp* (on *setjmp*(3C)), and other functions that do not return are not understood; this causes various inaccuracies.

SEE ALSO

cc(1), cpp(1), make(1).

Lint C Program Checker, tutorial in *C Programming Tools* manual.

STANDARDS CONFORMANCE

lint: SVID2, XPG2, XPG3

NAME

protogen – ANSI C function prototype generator

SYNOPSIS

protogen [*options*] *files*

DESCRIPTION

protogen is an experimental tool that helps convert old style C code to ANSI C by generating prototypes for function declarations. *protogen* does not expand macros or remove **#ifdef** preprocessor directives; it only alters the declarations of functions. This approach retains the original form and content as much as possible. However, it also imposes restrictions on the file being parsed. The restrictions are:

- The program compiles with no errors.
- Macro expansions cannot contain braces, parentheses, or semicolons. *protogen* uses these symbols to determine where functions begin and end. For example, a program that uses the following macros will not be prototyped correctly.

  ```
  #define BEGIN {
  #define END   }
  ```

- No preprocessor directives are allowed within a function declaration. For example, the following function will not be prototyped and a warning will be issued.

  ```
  foo(a,b
  #ifdef EXTENDPARAMS
  ,c,d
  #endif
  )
  char a,b
  #ifdef EXTENDPARAMS
  ,c,d
  #endif
  ;
  ```

- No more than one macro is used as a type specifier in a declaration. For example, the following function will not be prototyped.

  ```
  #define EXTERNTYPE extern int
  #define SPECIALTYPE register auto
  EXTERNTYPE SPECIALTYPE foo(){}
  ```

- Programs that use erratic macro substitutions will not be prototyped correctly. For example, when prototyping the following:

  ```
  #define X int foo
  #define Y a,b
  X(Y){ ...
  ```

  protogen produces:

  ```
  int X(int Y){...
  ```

Code that violates any of these assumptions can be preprocessed by *cc*(1) using the **–P** option before being prototyped. Use of *protogen* on such code may cause the offending function to be ignored and no prototype generated, the issuance of a grammar conflict message or other warning, the incorrect prototype to be generated, or, when extremely erratic macro substitutions are used, *protogen* may produce unpredictable results. In any case, *protogen* does not alter the original C source file.

When invoking *protogen*, use the desired options and then a list of files to be prototyped. *protogen* then prototypes each file individually, and appends all prototypes to a common file specified by the **–h** option or to the default file **prototypes.h**. All files containing functions that do not have prototypes result in the creation of a new file with the functions prototyped. Thus, if the following files need new prototype declarations, their resulting modifications would appear under the result file name in the current directory.

Source File		Result File
filename.c	→	filename.a.c

filename.h → filename.a.h
filename → filename.a.c

If *protogen* encounters an **#include** directive and the file name is enclosed in angle brackets it is considered a system file and no modified result file is produced. The file is scanned for **#defines** and **#ifdef** directives. If, however, the file name is enclosed in double quotes, it is scanned for prototype generation. If new prototypes are added, the modifications are produced in a result file following the same naming conventions.

Once the new result files have been created, each should be compared to the original using the *diff*(1) command. Manual inspection of each file should show that only function declarations have been altered. If any of the above assumptions where violated, they show up in the results of the *diff* operation. Next the **prototypes.h** file should be included in the proper location of each result file and compiled using the ANSI C compiler. Once all files have been checked and compile successfully, they should be renamed and passed through *protogen* if other functions inside **#if** sections still exist.

Options

protogen recognizes the following options:

−D *name*=*def*
−D *name* Defines name as if by a **#define** preprocessor directive (see *cpp*(1)).

−U *name* Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular processor. See *cpp*(1) for a list of possible predefined symbols.

−I *dir* Changes the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the standard directories. See *cpp*(1) for more information on the **−I** option.

−w Forces all prototype definitions to be widened to the standard C default promotions. All char, unsigned char, short, and unsigned short declarations are promoted to int. All floats are converted to double.

−h *file* Specifies header output file for prototypes. If this option is not given, the default output file **prototypes.h** is used.

EXAMPLES

Let this be a sample C program to be prototyped, where NODE, **RecordTypeHP300**, and **RecordTypeOther** are defined in **def.h**.

```
/******************************/
/* File: sample.c            */
/******************************/
#include "def.h"
#ifdef HP300
NODE
*foo(a,b,c)
RecordTypeHP300 c;
#else
NODE
*foo(a,b,c)
RecordTypeOther c;
#endif
{/* function statements */}

foo2(a,b)
long a;
char *b;
{/* function statements */}
```

Since there are two possible paths through the program, depending on the existence of the define HP300, *protogen* requires two passes to prototype the program.

```
$protogen -D HP300 -h sample.h sample.c
prototyping sample.c
no change   def.h
generating  sample.a.c
```

protogen has now prototyped the first declaration of **foo** and **foo2**. The results are placed in **sample.a.c**. The source file and result file should then be compared using the *diff* command and manually inspected. After checking the contents of the file, it should be moved to a temporary file and prototyped for the second path.

```
$mv sample.a.c temp.c
$protogen -U HP300 -h sample.h temp.c
prototyping temp.c
no change   def.h
generating  temp.a.c
```

The contents of temp.a.c:

```
/******************************/
/* File: sample.c            */
/******************************/
#include "def.h"
#ifdef HP300
NODE
*foo(int
a,
            int b,
            RecordTypeHP300 c )
#else
NODE
*foo(int
a,
            int b,
            RecordTypeOther c )
#endif
            {/* function statements */}
int foo2(long a ,
            char *b )
            {/* function statements */}
```

After inspecting the file for differences, the prototype header file should be inserted into the apropiate location of the source file. Line 4 would be the proper position for it in **temp.a.c**. Next, look at the header file.

The contents of sample.h:

```
#if defined(HP300)
/*
Options  -DHP300
*/

NODE *foo(int a,
            int b,
            RecordTypeHP300 c );

int foo2(long a ,
            char *b );

#endif

#if !defined(HP300)
/*
Options  -UHP300
```

```
*/
NODE *foo(int a,
       int b,
       RecordTypeOther c );

int foo2(long a ,
           char *b );

#endif
```

Note that the include file has two sections for each run. Only the section with the matching defines is used. This allows easy conversion of programs that require several passes through *protogen*. However, it also causes multiple declarations of the same function as with **foo2()**. Therefore, it might be more desirable to place the function prototypes in manually.

Once the prototypes are in place, try to compile the new source with the ANSI C compiler. When the program compiles and executes properly, back up the old source and replace it with the new ANSI source code.

Common Problems When adding Prototypes

For large programs, a common include file of all the prototypes decreases abstraction between files and may introduce naming conflicts. Many definitions and types may also have to be added to the include file in order for the prototypes to be valid. It is therefore very important to prototype programs together only if they make several calls to each other or in closely related groups. Use *protogen* to generate special prototype header files for programs that are in different groups but make frequent calls to a common group of functions. In most cases it is desirable to manually replace all empty declarations with their new prototypes and not introduce any new header files.

The **−w** option can be used to widen parameters to their default promotions in function declarations. It is necessary to remove empty declarations for functions that do not use default promotion types and were not prototyped with the **−w** option. If any such declarations do exist or they use typedefs that contain nonstandard promoted types, the following error message from *cc*(1) will be issued:

```
function prototype for <func-name> must contain
parameters compatible with default argument
promotions when used with an empty declaration
```

protogen is simply a scanner and performs no semantical analysis. It does not look at typedefs and does not alter them when given the **−w** widen option. Typedefs that need to be widened, must be done manually.

Functions that have variable arguments must be fixed manually, using the ellipsis notation.

DIAGNOSTICS

Error messages are sent to standard error, and report warnings and grammar conflicts. Warnings are intended to be self-explanatory. Grammar conflicts occur due to violations in the predefined assumptions *protogen* was designed for. If a grammar conflict occurs, try preprocessing.

SEE ALSO

cpp(1), cc(1).

Index

A

Aligning structures, 2-14
Alignment mode
  DOMAIN_NATURAL, 2-16
  DOMAIN_WORD, 2-16
  HPUX_NATURAL, 2-16
  HPUX_NATURAL_S500, 2-16
  HPUX_WORD, 2-16
  NATURAL, 2-16
ANSI C, 5-1
ANSI C, name space, 5-13
ANSI C, silent changes, 5-14
arrays, 2-8

B

bit field declared without signed keyword, 5-14
bit field declared without unsigned keyword, 5-14
block scope, 5-16

C

C, 7.0 implementation dependencies, 4-1
C, 8.0 implementation dependencies, 4-1
char, 2-3
coloring register allocation, 3-24
common subexpression elimination, 3-25
compiler directives, 3-5
compiling a C program, 1-2
compiling an ANSI C program, 1-2
const, 5-2, 5-8
constant folding, 3-23
constant propagation, 3-23
constants, optimizations performed on, 3-23
copy propagation, 3-26

D

data type alignments, 2-4
data types, 2-1
dead code elimination, 3-24
dead store elimination, 3-26
DOMAIN_NATURAL, 2-4, 2-16
DOMAIN_WORD, 2-4, 2-16
double, 2-3

E

enum, 2-3
error messages during level 2 optimization, 3-14
expressions, optimizations performed on, 3-23, 3-25, 3-26

F

filler bytes, 2-10
float, 2-3
floating point expressions with float parameters, 5-16
floating point precision, possible problems, 3-19
function prototype, 5-4

function prototype advantages, 5-4

G

global variables, 3-19
global variables and optimization, 5-3
global variables, effects on optimization, 3-10

H

HP_ALIGN, 2-14
HP_INLINE_DEFAULT directive, 3-9
HP_INLINE_FORCE directive, 3-7
HP_INLINE_LINES directive, 3-6
HP_INLINE_NOCODE directive, 3-10
HP_INLINE_OMIT directive, 3-8
HP_SHLIB_VERSION, 4-2
HP_SHLIB_VERSION, version pragma, 4-2
HP-UX, name space, 5-13
HPUX_NATURAL, 2-4
HPUX_NATURAL alignment mode, 2-16
HPUX_NATURAL_S500, 2-4
HPUX_NATURAL_S500 alignment mode, 2-16
HPUX_WORD, 2-4
HPUX_WORD alignment mode, 2-16

I

implementation dependencies
  7.0, 4-1
  8.0, 4-1
initializing variables, 3-19
int, 2-3
integer constants, unsuffixed, 5-16
internal errors during compilation, 3-16

L

level 2 optimization, compiler messages during, 3-14
library routines, naming conventions, 4-1
lint, 3-19
long, 2-3
long double, 2-3
loop induction variables, 3-30
loop invariants, 3-28
loops, optimizations performed on, 3-27
loop unrolling, 3-27

M

maxdsiz kernel configuration parameter, 3-18
memory allocation, optimizations performed on, 3-26
memory-mapped input/output, 3-19
memory, out of memory error during compilation, 3-17

N

name space, 5-11
NATURAL, 2-16
NATURAL data types, 2-6
NO_SIDE_EFFECTS directive, 3-10

O

+O1 compiler option, 3-4
+O2 compiler option, 3-4
-O compiler option, 3-4
optimization, 3-20
  assumptions, 3-19
  global variables, 3-10
  problems encountered during, 3-19
  troubleshooting, 3-19
  volatile, 3-4, 5-3
  warning messages, 3-14
optimization compiler options, 3-4
optimization levels, OPT_LEVEL, 3-6
optimization levels overview, 3-2
optimizations
  transformations, 3-20
optimization transformations
  coloring register allocation, 3-24

common subexpression elimination, 3-25
constant folding, 3-23
constant propagation, 3-23
copy propagation, 3-26
dead code elimination, 3-24
dead store elimination, 3-26
loop unrolling, 3-27
peephole optimization, 3-21
OPTIMIZE directive, 3-5
OPT_LEVEL directive, 3-6
out of memory error during compilation, 3-17
+OV compiler option, 3-4, 3-19

P

padding bytes, 2-10
peephole optimization, 3-21
pointer, 2-3
position independent code, 4-2
position independent code (PIC), 4-2
POSIX, name space, 5-13
pragma, 3-5
promotion rules for unsigned char, 5-15
promotion rules for unsigned short, 5-15
protogen, 5-6

Q

qualifiers, const and volatile, 5-1

R

registers, optimizations performed with, 3-24
register storage class, 3-25

S

Series 500 data types, 2-6
Series 600/700/800 data types, 2-6
shared libraries, 4-2
shared libraries, support, 4-2
short, 2-3
signal handlers, 3-19
signed char, 2-3
signed int, 2-3
signed short, 2-3
structures, 2-9
swap space, increasing to accommodate optimizer requirements, 3-18

T

tail recursion, elimination, 3-31
trigraphs, 5-14

U

unreachable code, optimizations performed on, 3-24
unsigned char, 2-3
unsigned int, 2-3
unsigned long, 2-3
unsigned short, 2-3

V

variables, initializing, 3-19
version pragma, 4-2
volatile, 3-4, 3-19, 5-3, 5-8

X

XOPEN, name space, 5-13

HEWLETT PACKARD

Copyright ©1991
Hewlett-Packard Company
Printed in USA E0191

Manual Part No.
B1864-90008

Manufacturing Part No. B1864-90008

B1864-90008