HEWLETT® PACKARD

HP 9000 Computers

HP-UX Portability Guide

HP-UX Portability Guide

HP 9000 Computers

HEWLETT
PACKARD

HP Part No. B2355-90025
Printed in USA 8/92

Second Edition
E0892

Printing History

New editions are complete revisions of the manual. Update packages may be issued between editions.

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

First Edition	January 1991
Second Edition	August 1992

This edition replaces the *HP-UX Portability Guide*, B1864-90006, Edition 1. That edition was written to reflect changes to languages as of the HP-UX 8.0 languages release. This edition reflects changes made for the HP-UX 9.0 release.

Preface

Manual Contents

This manual is organized into the following chapters and appendices:

Chapter 1	*An Introduction to Portability* This chapter introduces you to the subject matter of this manual.
Chapter 2	*Writing Portable Programs* Provides general guidelines for writing portable programs. Read this if you want to understand potential porting pitfalls and how using industry standards can enhance portability.
Chapter 3	*Porting between Series 300/400 and 700/800* Because the Series 300/400 and 700/800 architectures are different, they cannot be completely compatible. This chapter describes the differences between Series 300/400 and 700/800 implementations that might cause problems when porting code between systems—for example, floating-point hardware differences.
Chapter 4	*Porting from BSD4.3 to HP-UX* Describes `libc`, `libm`, `libmp`, and `libU77` routines that are supported in BSD4.3 that may or may not be supported in HP-UX.
Chapter 5	*Porting C Programs* Describes general considerations for writing portable HP C programs, and shows how to port between the following: ■ Series 300/400 and 700/800 ■ traditional C and ANSI C ■ HP C and Domain/C ■ HP C and VMS C It also describes how to call routines written in languages other than C.

Chapter 6	*Porting FORTRAN Programs* Describes general considerations for writing portable HP FORTRAN programs, and how to port between the following systems: ■ Series 300/400 and 700/800 ■ HP-UX FORTRAN and VMS FORTRAN It also describes how to call routines written in languages other than FORTRAN.
Chapter 7	*Porting Pascal Programs* Describes general considerations for writing portable HP Pascal programs, and how to port between the following systems: ■ Series 300/400 and 700/800 ■ HP-UX Pascal and the HP Pascal Workstation Also describes how to call routines written in languages other than Pascal.

Additional Documentation

The following manuals are referenced in this manual:

- *HP-UX Reference* (all Series) (B2355-90033)
- *Procedure Calling Conventions Reference Manual* (Series 700/800) (09740-90015)
- *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* (Series 700/800) (09740-90039)
- *Precision Architecture and Instruction Set Reference Manual* (Series 700/800) (09740-90014)
- *Assembly Language Reference Manual (Series 700/800)* (92432-90001)
- *How HP-UX Works: Concepts for the System Administrator* (all Series) (B2355-90029)
- *HP-UX System Administration Tasks* (Series 700/800) (B3108-90005)
- *HP-UX Assembler Reference and Supporting Documents* (Series 300/400) (B1864-90014)
- *Shells: User's Guide* (all Series) (B2355-90046)
- *Programming on HP-UX* (all Series) (B2355-90026)
- *C Programmer's Guide* (Series 300/400) (B1864-90008)

- *HP C Programmer's Guide* (Series 700/800) (92434-90002)
- *HP C/HP-UX Reference Manual* (Series 700/800) (92453-90024)
- *HP-UX Floating-Point Guide* (Series 700/800) (B2355-90024)
- *HP Pascal/HP-UX Programmer's Guide* (Series 700/800) (92431-90006)
- *HP Pascal Reference (Series 300/400)* (B2373-90000)
- Pascal Workstation documentation set:
  - *Pascal Workstation System Volumes I & II* (98615-90023)
  - *Pascal Procedure Library* (98615-90032)
  - *Pascal Graphics Techniques* (98615-90037)
  - *Pascal User's Guide* (98615-90042)
- *FORTRAN/9000 Programmer's Reference* (all Series) (B2408-90010)
- *FORTRAN/9000 Programmer's Guide* (all Series) (B2408-90009)

Conventions

Typeface Conventions Used throughout Manual

If You See This . . .	**It Represents . . .**
`typewriter font`	Computer literal text—for example, messages displayed by the computer, names of files and directories, keywords and identifiers in programming languages.

<u>`underlined text`</u>	Text that you type in examples. For instance, in the example below, you would type the underlined text (the `ls` command) to see the files in the current directory:
	`$ ls`
	`bar      foo`

bold text	Newly introduced terms in the text. For example, "a **library** is a collection of often-used routines that can be linked with object files to create executable programs."
italic text	Depending on the context, italic text can represent any of the following: ■ Emphasis—as in "do *not* press this button." ■ Book titles—as in "refer to *HP-UX Portability Guide*." ■ Pages in the *HP-UX Reference*—for example, "see *cc*(1)" means to refer to the "cc(1)" page in section 1 of the *HP-UX Reference*. ■ Variable text that you must type in place of the italics—as in "type `cc` *filename*." In this case, you would type a valid file name in place of *filename*.

Contents

Figures

Tables

1

An Introduction to Portability

What is Portability?

The process of porting a program consists of making any necessary modifications to enable it to run on a different computer, or, in some cases, on the same computer with a different or updated operating system. As a result of a hardware and/or operating system upgrade, existing applications must often be moved to the new system.

If the programmer uses inherently portable code as he or she develops new programs, considerable time and expense can be saved in the long run if porting becomes required. If, on the other hand, the programmer, for whatever reason, does not have the luxury of planning for or having portability already built into programs, he or she may be faced with making a substantial investment of time to convert the code after the fact.

Note The term "porting" is sometimes used interchangeably with "migrating" within computer documentation or common usage. This manual will use the term "porting" exclusively since its focus is on porting programs while "migrating" sometimes suggests other user or systems activities as well.

What This Manual Covers

This manual presents guidelines for both writing inherently portable code and for converting an existing program after the fact. Most of the information on writing portable code is presented in Chapter 2, "Writing Portable Programs". Other information on writing portable programs is presented in Chapters 5, 6, and 7 under the heading "General Portability Considerations". The remainder of the information in the manual is geared toward programmers who must belatedly do porting tasks from one platform or operating system or operating system version to another.

This manual deals primarily with porting programs written in C, FORTRAN, and Pascal. Further, it is restricted to a discussion of porting issues between any of the various HP 9000 implementations and between an HP 9000 system and another vendor's system.

In particular, it provides useful information to programmers regarding each of the following porting scenarios:

- porting programs between Series 300/400 and Series 700/800
- porting programs between HP-UX and BSD 4.3
- porting programs between HP-UX and VMS™
- porting programs between HP-UX and Domain/OS
- porting programs between HP-UX and the Pascal Workstation

This manual also describes how to call routines written in another languages from within your HP-UX program—for example, how to call C routines from FORTRAN.

What This Manual Does Not Cover

This manual is intended for programmers. As such, it does not cover general differences between the particular operating systems discussed that might be of interest to general users or system administrators, or that might be of value to those considering an overall migration from one system to another. The focus is on programming information, system calls, and library information only.

Because portability information on each of the following HP-UX languages is included in existing documentation sets for each language, this manual does *not* cover:

- Ada
- COBOL

It also does not cover porting issues that arise if you must rewrite your program from one language to another—for example, converting a FORTRAN program to a C program.

2

Writing Portable Programs

This chapter presents general guidelines for writing portable programs. It first describes a philosophy of portability—that is, the predisposition a programmer should have to write code that can be more easily ported between different systems. Then it describes some specific common causes of portability problems. Finally, it describes the UNIX standards and language standards that HP adheres to, the use of which promotes portable code.

In sum, this chapter describes

- a philosophy of portability
- common portability problems
- isolating system-specific code
- the use of industry standards

A Philosophy of Portability

Your approach to portability should be straightforward: Avoid using language or operating system features that are found only on a particular system; try to use only standard language and operating system features. In general, use programming techniques that minimize or avoid the use of system-dependent features. Following this philosophy will make it easier to port programs between different systems.

This does not mean that you should never use non-standard language features. Some non-standard features may provide great increases in performance or productivity. Nevertheless, before using a non-standard language feature, ask whether the benefits of using the feature outweigh the potential disadvantages if you must port the code to another system that does not support the feature. It is not always an easy decision, and it must be weighed carefully.

Common Causes of Portability Problems

This section describes some of the common causes of portability problems.

Non-Standard Language Extensions

Using industry standards is crucial to ensuring that your code is portable. Code that uses only standard features is more easily ported to other systems that support the same standard. The section "Industry Standards" at the end of this chapter summarizes various industry standards that HP is committed to following.

The HP-UX compilers have compile line options (command options) that make the compilers flag the use of non-portable constructs in your programs. Table 2-1 summarizes these options for each language. For details on these options, see the appropriate page in the *HP-UX Reference*.

Table 2-1. Standard-Enforcing Compile Line Options

Language	Option	Description
C	`-A`*s*	The parameter *s* can be the letter `c` or `a`. If it is `c` (`-Ac`), the compiler enforces the C language as defined by Kernighan and Ritchie's *The C Programming Language*, First Edition, sometimes referred to as K&R C. It also includes some Berkeley Software Distribution (BSD) extensions. If it is `a` (`-Aa`), the compiler enforces the ANSI X3.159-1989 (ANSI C) standard. If this option is not specified, the compiler assumes `-Ac` option. (See *cc*(1).) For further information, refer to "Porting Between K&R C and ANSI C" in chapter 5.
FORTRAN	`-a`	Produces warning messages for any non-ANSI 77 features, but the program still compiles. (See *f77*(1).)
	`-A`*s*	Allows you to choose which non-standard features cause the compiler to generate warning messages. (See *f77*(1).)
Pascal	`-A`	Produces error messages for any non-ANSI Pascal features. (See *pc*(1).)

Unstructured Programming

Structured programs are easier to understand. A well designed program that is modular is inherently more portable.

To this end, HP-UX provides tools that can aid in finding unstructured program constructs—for example, uninitialized variables or unreachable code. For the C programming language, the `lint` program can be used; for FORTRAN, the `lintfor` program is useful. Since these programs often produce large quantities of warning messages to `stderr`, it is best to capture their output by redirecting it to a file for later viewing.

Note also that these commands do not produce an object file—they only check program structure.

Compile Line Options and Compiler Directives

Compile line options that flag the use of non-standard features, such as those shown in Table 2-1, are clearly useful to programmers who want to write portable code. However, some options and directives enable system-dependent features and extensions that decrease the portability of code. Be sure to review the use of options and directives when porting code between systems because it is unlikely that the systems you port to will support all the same directives.

Assembly Language Code

Because assembly language code is based on the architecture of the machine on which it runs, assembly language code is not usually highly portable between systems. When the assembly language routines are ported to a system that has a different architecture, the assembly language will almost certainly need rewriting. And assembly language is one of the most difficult languages to work in for most programmers. Fortunately, with the optimization technologies available on many compilers today, programming in assembly language is less of a necessity than it used to be.

Absolute Addressing

Absolute addressing is the programming practice of using numeric constants to refer to objects by their absolute virtual or physical memory addresses as opposed to referring to symbolic addresses. Absolute addressing is sometimes done to take advantage of knowledge about the location of various objects in virtual memory. The problem is that such knowledge is highly system-dependent, and, therefore, tends not to be portable. Note that absolute addressing is not the same as incrementing or decrementing pointers.

Language Semantics

Because a programming language defines a program's meaning, differences in semantics for a given language between different systems can affect portability. Unless such semantics are completely identical, you may find that a program written in that language will produce different results on the two systems. Unfortunately, it is often difficult to know beforehand whether the semantics of a compiler implementation for one language will be the same on another system. To help you to be aware of potential differences, later chapters on porting C, FORTRAN, and Pascal summarize some of the languages' features that may have different semantics on other systems.

Floating-Point Fuzziness

Floating-point operations can complicate compatibility. Computer floating-point values only closely approximate actual numbers, so when comparing floating-point values, it is best to compare to a range of values instead of a single value. This technique is known as a "fuzzy compare." For example, in a fragment of Pascal code, you could replace

```
if (x = 1.2267) then
     y:= y + 1;
```

with a more accommodating fragment of code such as:

```
if (abs(x - 1.2267) < err_margin) then
     y:= y + 1;
```

for comparisons. The value of `err_margin` will be very small; however, it will *not* be constant across all HP-UX implementations. For example, it will differ among Series 300/400 systems, depending on whether the program was compiled with the `+ffpa` or `-O` option.

For more information about how floating-point operations can affect compatibility, refer to the *HP-UX Floating-Point Guide*.

Data File Incompatibility and Input/Output

File manipulation and input/output operations have traditionally been two of the most troublesome areas impacting portability. Most language standards are intentionally vague in these areas to allow vendors to make the most effective use of their architectures. Unfortunately, file manipulation and input/output operations are also frequently critical to performance, so they are usually manipulated in a system-dependent manner. The apparently conflicting goals of portability and performance can be met by a careful design that encapsulates interface routines.

Additionally, input/output operations should be performed in the same language as the main program; for example, avoid having a main FORTRAN program that calls C input/output routines. Methods exist to do input/output from external routines, but they are not generic. In addition, difficult problems can be encountered if input/output is performed from more than one language at a time since each language has its own buffers.

Data Alignment Differences

Data alignment refers to the way in which a system or language aligns data structures in virtual memory. For example, by default Series 300/400 C aligns variables of type `double` on 4-byte boundaries, while Series 700/800 C aligns them on 8-byte boundaries. This is done to realize greater efficiency in accessing data based on a particular hardware architecture.

These differences in alignment can cause problems for code that makes assumptions about the location of variables. It can also cause problems for programs that write and read data structures to files: A file written by one system may be read incorrectly by the same code on another system because the variable alignment is different on the two systems.

Data type alignments for each language are summarized in the appropriate language chapters later in this manual. In addition, HP-UX provides ways to force a particular alignment on different systems; for example, to force Series 700/800 C to align data the same way as Series 300/400, use the `#pragma HP_ALIGN HPUX_WORD` directive. These directives are covered in later language-specific chapters.

Isolating System-Dependent Code

It is not always possible to avoid using some non-standard features. It may be that you can get better performance from using some non-standard features, or may be able to accomplish some feat that isn't possible with standard features. In such cases, it is best to "isolate" such system-dependent code in include files, libraries, or conditional compilation blocks. Then, when you port the code to another system it will be clear what code must be rewritten for the new system.

Industry Standards

The use of industry standards for UNIX and languages is crucial to portability.

UNIX Standards

Hewlett-Packard Company adheres to UNIX standards in the following ways:

- HP-UX is based on UNIX System V.3. All HP-UX implementations pass SVVS validation.
- HP-UX has added selected BSD4.2 and BSD4.3 extensions that have become *de facto* industry standards.
- HP-UX itself is an internal corporate standard that has been designed to maximize portability across the HP 9000 product family, regardless of architecture. The HP-UX standard concerns itself with both software and documentation.
- Hewlett-Packard is an active participant in the developing POSIX standard. HP intends to make HP-UX track this standard.
- Likewise, Hewlett-Packard is committed to following the X/OPEN standard.

HP-UX and Multiple Standards

Industry standards for UNIX overlap and at times even conflict. In order to support portability of applications that conform to different standards, HP-UX provides multiple interfaces to selected services and enables administrators to configure certain aspects of the run-time environment.

- When routine names do not conflict, `libc` contains overlapping routines, such as `bcopy` from BSD and `memcpy/memmove` from ANSI C, X/Open, and SYS V.
- Compatibility libraries may be provided (for example, see *bsdproc*(2)).
- File systems can support either 14-character filenames or long filenames (see *convertfs*(1M)).
- Administrators can control the run-time environment via environment variables or `setprivgrp()`.

In some cases routines are provided for portability, but not all routines have equivalent performance.

Note The use of POSIX signals is strongly recommended for maximum performance and portability.

Language Standards

Each language described in this manual is subject to industry standards. Information on standards for C++ is also presented below.

Standards for C

All HP-UX systems support compilation in two modes: compatibility mode and ANSI mode.

Compatibility mode which is the default supports the C syntax and semantics of previous releases in order to provide full backward compatibility with C code written prior to the Series 800 HP-UX 7.0 release and Series 300/400 7.40 release. Although Series 300/400 and 700/800 are not fully compatible, there is a high degree of compatibility between the two implementations, as described in Chapter 5.

Beginning with release 7.0 on Series 800, release 7.40 on Series 300/400, and release 8.05 on Series 700, ANSI mode became available. This mode provides a full implementation of ANSI X3.159-1989. The HP implementation of ANSI C also conforms to FIPS 160 and ISO 9899:1990. Of all HP-UX languages, C has the reputation as being the most portable, due to its high adherence to standards.

Standards for FORTRAN

FORTRAN, being one of the oldest high level programming languages, has a long history of standardization. The most widely accepted current standard is ANSI X3.9-1978, commonly known as FORTRAN 77. All HP-UX FORTRAN compilers fully comply with this standard and have been federally validated. A common set of extensions is set forth in the U.S. Department of Defense publication, *MIL-STD-1753 Military Standard FORTRAN, DOD Supplement to American National Standard X3.9-1978*. These extensions have been fully implemented. HP-UX FORTRAN on all implementations also conforms to FIPS PUB 69-1.

Standards for Pascal

The most widely recognized standard for Pascal is ISO 7185-1983. ANSI 770X3.97-1983 is nearly identical to level 0 of this standard. HP Pascal is a superset of ISO 7185-1983 level 0 and and a superset of level 1 with minor exceptions. HP also has an internal corporate standard to which Series 300/400 and 700/800 implementations conform or are converging. Pascal on these architectures conforms to ISO 7185-1983 level 0 at present.

Standards for C++

HP C++ is based on USL's implementation of `cfront`, the C++ to C translator. HP C++ is, however, a true compiler, compiling C++ source code directly to object code. HP C++ conforms to the definition of C++ as described in the *The Annotated C++ Reference Manual*, which is being used as the base document by the ANSI C++ standards committee.

3 Porting between Series 300/400 and 700/800

Although Series 300/400 and Series 700/800 are highly compatible, some differences are inevitable because of architectural differences. Specifically, Series 300/400 is Motorola-based and Series 700/800 is PA-RISC-based.

This chapter summarizes some of the differences between the implementations which you should be aware of when porting code between them. Specifically, this chapter discusses

- system architecture differences
- identifying the system at run time
- determining the processor at run time (on Series 300/400)
- differences in object files and associated tools
- optimization differences
- language differences
- system call and library differences
- floating-point hardware

System Architecture Differences

Series 300/400 computers are based on the Motorola 68000 (68K) series of microprocessors and co-processors. Most Series 300/400's running HP-UX have either a 68020 processor with a 68881 floating-point co-processor, a 68030 processor with a 68882 floating-point co-processor, or a 68040 processor with a built-in co-processor. (Refer to Table 3-2 later in this chapter for more complete information.)

Series 700/800 computers are based on PA-RISC architecture. Series 700 computers are based on PA-RISC Version 1.1 which is an upward-compatible evolution of PA-RISC Version 1.0. Some models of Series 800 computers are based on PA-RISC 1.0 and some on PA-RISC 1.1. Table 3-1 shows which Series 800 computers are PA-RISC 1.0 and which are PA-RISC 1.1. You can use the command `uname -m` if you do not know the model number of your system.

Table 3-1. Series 800 PA-RISC Architecture

PA-RISC 1.0	PA-RISC 1.1
808, 810, 815, 825, 835, 840, 845, 850, 855, 860, 865, 870	807, 817, 827, 837, 847, 857, 867, 877

Code compiled on the PA-RISC 1.1 machines will not execute on the PA-RISC 1.0 Series 800 machines unless it was compiled with the `+DA1.0` compiler option. It will execute, however, on other PA-RISC 1.1 machines.

If you do not specify the `+DA` compiler option, `DA1.1` is the default on the Series 700; `DA1.0` is the default on the Series 800. For best performance, you should specify `+DA` with the architecture or model number of the system where you plan to execute the program.

The `+DS` compiler option performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture. If you do not specify this option, the default instruction scheduling is for the system you are compiling on. To improve performance on a particular model of the HP 9000, use `+DS` with that model number. You should note that scheduling tuned for a particular model *will* still execute on other HP 9000 systems, although possibily less efficiently.

For more information on the +DA and +DS compiler options, see Chapters 5, 6, and 7.

Motorola-based and PA-RISC-based architectures have completely different instruction sets, so any assembly language routines will have to be converted by hand. Such a port is beyond the scope of this document.

Byte Ordering, Word Size, and Alignment

Both the Motorola-based and PA-RISC architectures have 32-bit words with the most significant byte of the word having the lowest address (used to access that word). However, PA-RISC has stricter requirements on how multi-byte data is aligned in memory. Data alignment is one of the biggest portability issues and is discussed in the subsequent chapters on C, FORTRAN, and Pascal.

Program Address Space

The layout of a process' logical address space is different for the two architectures. For most programs, the differences are unimportant. However, you should be aware of these differences if your programs do any of the following:

- Use shared memory
- Do custom memory management
- Generate executable code
- Access architecture-dependent memory addresses

The Series 300/400 has a linear four gigabyte address space, shown in Figure 3-1.

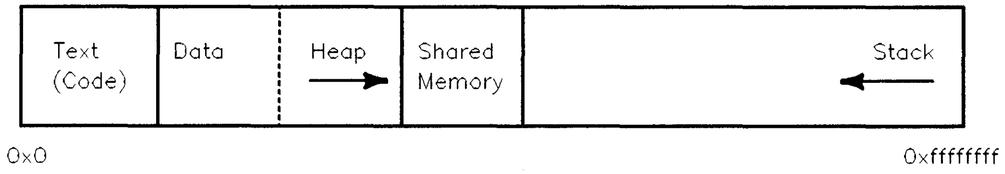

Figure 3-1. Series 300/400 Process Address Space

The layout of memory on the Series 700/800 is a little more complicated and in order to keep this description brief, a few details have been glossed over. The details are described in other documents, listed at the end of this section. Figure 3-2 shows what memory looks like on the Series 700/800 if we look at it as having a four gigabyte linear address space.

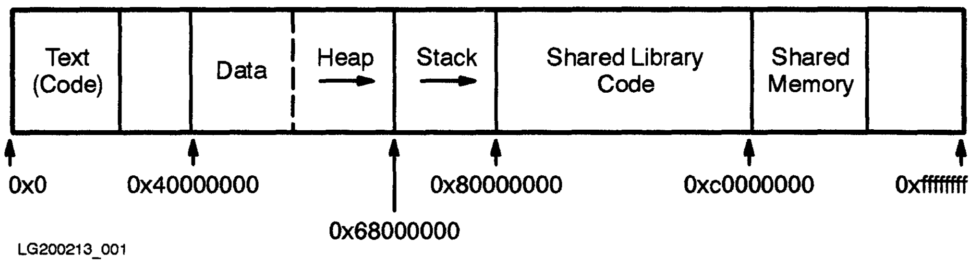

Figure 3-2. Series 700/800 Process Address Space

Note You should not rely on the boundaries shown above as they may be modified from release to release.

Here are some important implications of these address space differences:

- First, if your programs manipulate the most significant two or three bits of pointers because you think you will never have an address that high, your code will break on the Series 700/800 because data are allocated at addresses above 0x40000000.
- Use of shared memory is also slightly different. On the Series 700/800, you must let the system decide where to attach shared memory by setting the *shmaddr* parameter to 0 when calling **shmat** (see *shmat*(2)). Also, due to the way the Series 700/800 addresses shared memory through space registers, if you are accessing more than two shared memory segments, you may experience performance degradation, particularly if accesses to the different segments are interleaved.

If you need more detailed information on these types of issues than the short overview provided here, consult the following HP manuals:

- *Precision Architecture and Instruction Reference Manual* (Version 1.0, Series 700/800).
- *PA-RISC 1.1 Architecture and Instruction Reference Manual* (Version 1.1, Series 700/800).
- *Procedure Calling Conventions Manual* (Series 700/800).
- *Assembly Language Reference Manual* (Series 700/800).
- *How HP-UX Works: Concepts for the System Administrator* (all Series).
- *HP-UX Assembler Reference and Supporting Documents* (Series 300/400).

Memory Organization Differences

On HP-UX computers, memory is accessed by byte address. The most significant byte of a memory element is addressed first and has the lowest numeric address. This value is used to access the entire memory element. For example, a 32-bit integer is accessed by the most-significant byte address. This memory addressing system is referred to as big-Endien.

The conventions of bit numbering differ on HP-UX implementations, as shown in Figure 3-3. On Series 300/400, the most significant bit of a long word is bit 31; on the Series 700/800, the most significant bit is bit 0.

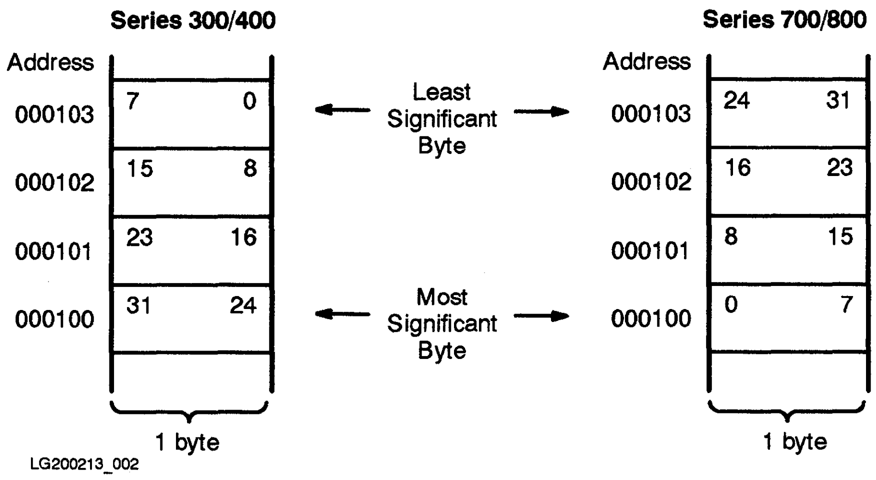

Figure 3-3. Series 300/400 and 700/800 Memory Organization Differences

Code and Data Size Limitations

On Series 300/400, the maximum address space is four gigabytes (4Gb). On Series 700/800, the code is limited to 1Gb, data is limited to 1Gb, and shared memory is limited to approximately 0.75Gb. For details, refer to *How HP-UX Works: Concepts for the System Administrator*.

Execution Stack and Parameter Lists

The execution stacks on the two architectures are also different: the stack on the Series 300/400 grows towards lower addresses; the stack on the Series 700/800 grows towards higher addresses. In addition, the compiler may choose to pass some parameters through registers instead of on the stack.

Normally this information is not important because parameter-passing is invisible to programs. Occasionally, though, programmers may try to take advantage of this information to pass variable numbers of arguments to a routine—for example, by manually stepping through the execution stack. Such practice is highly non-portable.

If you need more information on the execution stack on the Series 700/800, you should consult the *Procedure Calling Conventions Manual*; on Series 300/400, consult *HP-UX Assembler Reference and Supporting Documents*.

Identifying the System at Run Time

When writing programs to run on multiple systems, it is sometimes necessary to determine the system configuration at run time. The **uname** system call can be used to determine machine type and other pertinent information. (Similarly, it may be necessary to determine what floating-point hardware is available; this is described later in the section "Floating-Point Hardware".) The program shown in Figure 3-4 calls the **uname** system call to display hardware and software information.

Note Run-time characteristics can also be determined by using the sysconf system call.

```
#include <sys/utsname.h>
#include <stdio.h>

main()
{
  struct utsname un;

  if ( uname( &un ) == -1 ) /* if -1, an error occurred */
  {
      fprintf(stderr, "uname failed\n");
      exit(1);
  }
  /* no errors occurred, so display the fields */
  printf("System name:      %s\n", un.sysname);
  printf("Node name:        %s\n", un.nodename);
  printf("HP-UX Release:    %s\n", un.release);
  printf("HP-UX Version:    %s\n", un.version);
  printf("Machine:          %s\n", un.machine);
  printf("ID Number:        %s\n", un.__idnumber);
}
```

Figure 3-4. Identifying the System with the uname System Call

Shown below is the output from compiling and running this program on a Series 300 Model 360 running HP-UX 9.0:

```
$ cc -Ac -o myuname myuname.c     Compile it.
$ myuname                         Run it.
System name:        HP-UX
Node name:          node1
HP-UX Release:      9.0
HP-UX Version:      B
Machine:            9000/360
ID Number:
```

Shown below is the output from compiling and running this program on a Series 800 Model 840 running HP-UX 9.0:

```
$ cc -Ac -o myuname myuname.c     Compile it.
$ myuname                         Run it.
System name:        HP-UX
Node name:          node2
HP-UX Release:      A.B9.00
HP-UX Version:      B
Machine:            9000/840
ID Number:          25427968
```

For details on the information returned by the **uname** system call, see *uname*(2). Within shell scripts, the **uname** command accomplishes a similar function (see *uname*(1)).

The **uname** function is a C function but can be called from other languages as well. For details on calling C functions from FORTRAN or Pascal, see the appropriate language chapter later in this manual.

You can also use the **getcontext** system call if you wish to get hardware and system configuration information in string form. A **getcontext** command is also available. For details, see *getcontext*(2) and *getcontext*(1).

Determining the Processor at Run Time on Series 300/400

A Series 300/400 computer uses either the MC68020, MC68030, or MC68040 processor, depending on the model. In addition, some models support more than one processor; for example, a Model 375 uses either the MC68030 or MC68040. Because of this, it is not always possible to determine from the `uname` system call what processor a computer is using. You should use `sysconf` to tell them apart.

Differences in Object File Space Allocation and Format

On the Series 300/400, if a global data item is declared in two or more files, the size allocated for that data item in the object file by the linker is the size of the initialized data, even if the declared size is different elsewhere. Hence, programs that declare global variables inconsistently will have unreliable results. The Series 700/800 linker, on the other hand, adjusts the allocated space to fit the largest declaration.

In addition, the Series 700/800 object file format is considerably different than on the Series 300/400. The differences go beyond just the differences in the binary machine instructions (see *a.out*(4) for details). As a result, there are some differences in two commands that deal with object files shown below. (You may want to check the *HP-UX Reference* for further details.)

- *nm*(1): The two implementations have completely distinct sets of options and very different forms of output. If you have any scripts that expect `nm` output in the format of the Series 300/400 version, they will have to be modified for the Series 700/800. Also, `nm` on the 700/800 now supports `-p` to get output similar to that on the Series 300/400.
- *ld*(1): There are several differences in options.

Optimization Differences

All HP-UX C and FORTRAN compilers perform the optimizations that are most effective on the particular architecture of the system you are using.

The Series 300/400 C and FORTRAN compilers have an optional global optimization pass to the compilation path. If `-O` or `+O2` is specified on the compile command line, the global optimization pass is enabled. On the C and FORTRAN compilers, `+O3` enables the global optimizer and the procedure integrator.

Series 700/800 C, FORTRAN and Pascal compilers also have global optimization. `-O` on Series 700/800 is roughly equivalent to `-O` on Series 300/400 although specific optimization techniques may differ between the two machines.

Optimization directives are also available in C and FORTRAN, and may act differently on Series 300/400 versus 700/800. For details, refer to each language's documentation set.

Language Differences

In general, there is a very high degree of compatibility between Series 300/400 and 700/800 languages. However, because of the differences in architecture and the origins of the compilers on the two systems, there are differences that you should be aware of when porting between systems. These differences for C, FORTRAN, and Pascal are documented in later chapters of this Guide, one for each language.

HP C++ is based on USL's `cfront` translator on both the Series 300/400 and the Series 700/800.

System Call and Library Differences

This section shows the differences between system calls and subroutine libraries across Series 300/400 and 700/800 HP-UX implementations. If you need more information, refer to the **DEPENDENCIES** section of the particular routine's man page in the *HP-UX Reference*; system calls are documented in section 2 ; subroutine library entry points are documented in section 3.

On Series 700/800 only, many of the system calls have separate entry points suitable for calling from FORTRAN (described in the *FORTRAN* documentation).

There are differences in the way shared libraries are implemented across Series 300/400 and Series 700/800. Some of this information is available in *Programming on HP-UX*.

System Calls

`acct`	On Series 300/400, the system accounting routine ignores locks placed on the process accounting file; also, if the process accounting file reaches 5000 blocks, records for processes terminating after that point are lost.
`exec`	Series 700/800 supports shareable executable files and demand-loadable executable output files, both created with the `-n` linker option. Unshareable files (created with the `-N` linker option) are not supported on Series 700/800.
`gettimeofday`	Series 700/800 has a granularity of 1 microsecond. Series 300/400 has a granularity of 4 microseconds.
`ptrace`	The functionality of `ptrace` is dependent on the hardware and will probably not be portable.
`reboot`	Various dependencies exist.
`rtprio`	Series 800 dependency exists.
`select`	Some dependencies in supported devices and file types.
`shmctl`	Series 300/400 has differences in the handling of `EACCES`.

3

`shmop`	Various dependencies exist. On Series 700/800, programs cannot attach shared memory at a specific address.
`signal`	The codes for illegal instruction (`SIGILL`) and floating-point exception (`SIGFPE`) signals are different on Series 300/400 versus Series 700/800.
`sigspace`	On Series 300/400, kernel overhead is taken out of reserved space.
`sigstack`	System dependencies exist as a result of differences in how the stack grows on the two architectures.
`sigsuspend`	Series 300/400 only.

Subroutine Libraries

`blmode`,`blclose`, `blread`, `blget`, `blset`	Series 800 only.
`cachectl`	Series 300/400 only.
`clock`	Clock resolution is 20 milliseconds on Series 300/400, 10 milliseconds on Series 700/800.
`crt0`	Various dependencies exist.
`cvtnum`	Series 300/400 only. (3) library call|
`dial`, `undial`	Use on the Series 300/400 causes an implicit call to *alarm*(2).
`end`	On the Series 700/800, the linker defines a `_text_start` and `_data_start` symbol.
`gpio_get_status`, `gpio_set_ctl`	System dependencies exist. Not available on Series 700.

hpib_abort, hpib_bus_status, hpib_eoi_ctl, hpib_card_ppoll_resp, hpib_io, hpib_pass_ctl, hpib_rqst_srvce, hpib_send_cmnd, hpib_spoll, hpib_status_wait, hpib_wait_on_ppoll	System dependencies exist.
hpib_address_ctl, hpib_atn_ctl, hpib_parity_ctl	Series 300/400/700 only.
hppac	Series 800 only.
_INITIALIZER	Series 300/400 can have a default initializer; a Series 700/800 initializer must be declared explicitly.
io_burst, io_dma_ctl	Series 300/400/700 only.
io_lock, io_unlock	Series 300/400 only.
io_get_term_reason, io_on_interrupt, io_reset, io_speed_ctl, io_timeout_ctl	System dependencies exist.
is_hw_present	Series 300 only.
shl_definesym	Series 300/400 only.
shl_findsym	On Series 300/400, symbol names begin with an underscore (_); on Series 700/800, they do not.
shl_get	On Series 300/400, the name of the main program is not known; the filename a.out is used instead. The format of the descriptor differs across implementations.

`shl_gethandle`	The format of the shared library descriptor varies slightly across implementations.
`shl_getsymbols`	Series 300/400 only.
`shl_load`	On Series 700/800, the *address* argument is ignored. `BIND_RESTRICTED` and `DYNAMIC_PATH` modifiers are not supported on Series 300/400.
`_toupper`, `_tolower`	Series 300/400 does not define the results for non-ASCII arguments.

Floating-Point Hardware

This section describes different floating-point hardware configurations available on Series 300/400 and Series 700/800.

Series 300/400 Floating-Point Hardware

Table 3-2 shows the various Series 300/400 hardware configurations with floating-point math performance.

Table 3-2.
Series 300/400 Floating-Point Hardware Configurations

Series	Processor	Floating-Point Co-processor	Floating-Point Card
318	68020	68881	None
319	68020	68881	None
320	68020	68881	None
330	68020	68881	Optional HP 98248A
332	68030	68882	None
340	68030	68882	None
345	68040	built-in	None
	or		
	68030	68882	None
350	68020	68881	Optional HP 98248A
360	68030	68882	Optional HP 98248B
370	68030	68882	Optional HP 98248B
375	68040	built-in	None
	or		
	68030	68882	None
425S	68040	built-in	None

The Motorola 68020 and 68030 processors differ in their speed and internal architecture, but use the same instruction set. The Motorola 68040 processor has an additional instruction, `MOVE16`, described in the manual *HP-UX Assembler Reference and Supporting Documents*.

The Motorola 68881 and 68882 floating-point co-processors use a compatible instruction set. The 68040 processor has a built-in math co-processor that supports most of the primary 68881 or 68882 floating-point instruction set. However, some 6888*x* instructions, such as `SIN`, are not implemented on the 68040 processor; instead, the 68040 traps such instructions to library calls that emulate the instruction in software. Such "trapped" instructions can potentially execute slowly. Therefore, to speed up execution, the compilers do not emit these instructions but call the library routines directly, thus eliminating the hardware "trap" that can slow down execution.

Identifying Your Series 300/400 Floating-Point Configuration at Run Time

You can determine the floating-point hardware configuration of your Series 300/400 by accessing flags set automatically in `/lib/crt0.o` for C and Pascal, and `/lib/frt0.o` for FORTRAN. Each flag is declared in C to be an external short int (2 bytes). Table 3-3 summarizes these flags.

Table 3-3. Series 300/400 Floating-Point Hardware Flags

Flag	Set To	Means
`flag_soft`	non-zero	HP98635A not present
`flag_68881`	non-zero	68881 or 68882 present
`flag_fpa`	non-zero	HP98248A or HP98248B present

In `/lib/libc.a`, the following four functions are defined to interrogate these flags:

`is_68010_present` (this is an obsolete processor)
`is_68881_present` (MC68881 and MC68882 are synonymous here)
`is_98248A_present` (98248A and 98248B are synonymous here)
`is_98635A_present` (this is an obsolete card)

These functions return a four byte integer (`int`) value. If the hardware is present, the function returns a value of 1; otherwise, it returns 0.

Series 300/400 Compile Line Options

Table 3-4 lists compile line options that can be used to specify floating-point hardware on Series 300/400.

Table 3-4. Series 300/400 Floating-Point Compile Line Options

Option	Model	Generation of Calls for Floating-Point Math	Considerations
default	All 300/400s	inline co-processor instructions	
`+M`	All 300/400s	user-provided library calls if present; else, `libc.a` and `libm.a`	user-provided calls are usually slower
`+ffpa`	330 or 350 with HP98248A; 360 or 370 with HP98248B	HP98248A/B	aborts if no HP98248A/B present
`+bfpa`	330 or 350 with HP98248A; 360 or 370 with HP98248B	HP98248A/B, if present; else, default	will run on any Series 300/400 except 310, which is now obsoleted

Recommendations

The `+bfpa` option for the Series 300/400 C and FORTRAN compilers does provide additional flexibility and performance, but it also tends to increase the code size of statements involving floating-point arithmetic. The effect of code expansion varies widely. You may want a trial compilation of your actual program for the most accurate measure of code size.

It is often advantageous to compile only critical regions of your programs with these options and to compile the remainder of the program without any floating-point hardware support.

Series 700/800 Floating-Point Hardware

The Series 700 has floating-point support built into the CPU and therefore it does not require any external floating-point support. Additionally, some of the Series 800 systems have optional floating-point hardware. However, regardless of whether floating-point hardware is present on a given Series 800 system, floating-point instructions can still be executed. If the floating-point hardware is not present, floating-point instructions will actually be emulated by the system software. As a result, all Series 800 systems can execute floating-point instructions.

4

Porting from BSD4.3 to HP-UX

This chapter lists BSD4.3 library routines, header files, and applications that are either not supported in HP-UX or implemented differently on HP-UX.

This chapter does *not* describe fixes for these routines. It merely lists them so you will be aware of them when porting BSD4.3 applications to HP-UX.

libc Entry Points

The following BSD4.3 `libc` entry points are provided on HP-UX but *only* in the library `libBSD`. A program can call these routines if it is linked with `libBSD` (`-lBSD`).

killpg(2)	*getwd*(3)	*sigvec*(2)

4

The following BSD4.3 `libc` entry points are found in both the HP-UX `libc` and `libBSD`:

getpgrp(2)	*setpgrp*(2)	*signal*(2)

To get the BSD4.3 implementation of these routines, the following `cc` command line should be used:

```
$ cc bsdprog.c -lBSD
```

Or, link with `libBSD`, making sure that `libBSD` precedes the `-lc` option on the command line. For example:

```
$ ld /lib/crt0.o bsdprog.o -lBSD -lc
```

The BSD4.3 `libc` entry points listed below are not provided in HP-UX. If your application calls these routines, you must provide fixes. For instance, you could write your own versions of the routines to accomplish the same action and link with them. Or you could use `ifdef`s to isolate such code, and use alternative standard methods to do the same thing.

adjtime(2)	*getpriority*(2)	*re_comp*(3)	*setruid*(3)
alloca(3)[1]	*getrusage*(2)	*re_exec*(3)	*setstate*(3)
alphasort(3)	*getttyent*(3)	*remque*(3)	*setttyent*(3)
byteorder(3N)	*getttynam*(3)	*setbuffer*(3S)	*siginterrupt*(3)
closelog(3)	*initstate*(3)	*setegid*(3)	*srandom*(3)
comp(3)	*insque*(3)	*seteuid*(3)	*ualarm*(3)
endttyent(3)	*moncontrol*(3)	*sethostid*(2)	*usleep*(3)
errlist(3)	*monstartup*(3)	*setkey*(3)	*utimes*(2)
flock(2)	*network*(3N)	*setlinebuf*(3S)	*valloc*(3C)
ftime(3C)	*ns*(3N)	*setpriority*(2)	*vhangup*(2)
getdisk(3)	*ntohl*(3N)	*setpwfile*(3)	*vlimit*(3C)
getdtablesize(2)	*ntohs*(3N)	*setregid*(2)	*vtimes*(3C)
gethostid(2)	*psignal*(3)	*setreuid*(2)	
getpagesize(2)	*random*(3)	*setrgid*(3)	

1 Although `alloca` is provided in `/lib/libPW.a`, you should note that its use is largely discouraged, especially where performance and/or memory utilization are critical. It is inherently non-portable and most public domain implementations do not work when linked with optimized code. It only reclaims space when it is called a second time, not when the calling routine returns. When linking this function, you can use the following `cc` command to prevent picking up undesired functions from `libPW.a`: `cc [-lBSD] -lc -lPW -Wl,-a,archive`. Or, do so by using this `ld` command: `ld [-lBSD] -lc -lPW -a archive -lc`.

libm Entry Points

For HP-UX 9.0, the following BSD4.3 `libm` entry points are not supported on the Series 300/400 or on the PA-RISC 1.0 versions of the math library on the Series 700/800:

acosh(3M)	*cbrt*(3M)	*finite*(3M)	*rint*(3M)
asinh(3M)	*copysign*(3M)	*infnam*(3M)	*scalb*(3M)
atanh(3M)	*drem*(3M)	*log1p*(3M)	
cabs(3M)	*expm1*(3M)	*logb*(3M)	

See the *HP-UX Reference* and the *HP-UX Floating-Point Guide* for more information about these functions.

The PA-RISC 1.1 versions of the math library on the Series 700/800 support all of these routines except *expm1*(3M), *infnam*(3M), and *log1p*(3M). (The PA1.1 library is the default on the Series 700. The PA1.0 library is the default on the Series 800.)

Note PA-RISC 1.1 versions of the math library may not run on the Series 800.

libmp Entry Points

Listed below are entry points in the BSD4.3 library `libmp`, which provides support for multi-precision math functions; they are not supported on HP-UX. If your programs call these routines, you must provide fixes for them.

fmin(3M)
fmout(3M)
gcd(3M)
invert(3M)
itom(3M)
m_in(3M)
m_out(3M)
madd(3M)
mcmp(3M)
mdiv(3M)
min(3M)
mout(3M)
move(3M)
msqrt(3M)
msub(3M)
mult(3M)
omin(3M)
omout(3M)
pow(3M)
rpow(3M)
sdiv(3M)

libU77 Entry Points

`libU77` is the FORTRAN BSD4.3 system call library. It is new on HP-UX with the 9.0 release. It is available on all the HP 9000 implementations.

Listed below are the entry points for the library functions. These functions provide an interface from *f77* programs to the system. The `+U77` *f77* compiler option causes it to recognize and load the functions. For more information, see *intro*(3F).

access
alarm
chdir
chmod
ctime
dtime
etime
falloc
fdate
fgetc
fork
fputc
free
fseek
fstat
ftell
gerror
getarg
getc
getcwd
getenv
getgid
getlog
getpid
getuid
gmtime
hostnm
iargc
idate
ierrno
isatty
itime
kill
link
loc
lstat
ltime
malloc
perror
putc
qsort
rename
signal
sleep
stat
symlnk
system
tclose
time
topen
tread
trewin
tskipf
tstate
ttynam
twrite
unlink
wait

Header Files

The following BSD4.3 header files are not provided in HP-UX:

```
/usr/include/mp.h
/usr/include/struct.h
/usr/include/lastlog.h
/usr/include/pcc.h
/usr/include/ttyent.h
/usr/include/vfont.h
/usr/include/frame.h
```

Applications

A concerted effort to port *all* BSD commands has not been made, though many—such as `vi`, `csh`, `more`, etc—*are* available. HP-UX does *not* include the applications listed below, which may cause problems when porting to HP-UX. There are additional missing applications not noted here.

- `/etc/timed`
- `/usr/bin/talk`
- `/etc/renice`
- `lpr/lpd`—though `lp` provides BSD-style network spooling
- bibliography tools

Porting C Programs

HP-UX C is standard C. By default, the HP C compilers support the Kernighan and Ritchie language definition (as described in *The C Programming Language*, First Edition) as well as some BSD extensions. This standard version is referred to as **compatibility mode** throughout this manual.

If the `-Aa` option is specified, the compilers use the ANSI C standard language definition. Whenever practical, new code should be written to conform to ANSI specifications since Series 300/400 and 700/800 compilers are ANSI compliant and there is more extensive error checking in ANSI mode. ANSI C code is also likely to be more portable across other vendors' systems.

Even though HP C is standard, there are some features, especially in implementation-specific areas, where other vendors' C may vary from HP C. This chapter describes general portability considerations when programming in HP C. In addition, it describes portability considerations for specific versions of C. It also describes how to call other languages from C. Specifically, this chapter describes:

- general portability considerations when programming in HP C
- checking for standards compliance
- porting between Kernighan and Ritchie compliant C (referred to here as K&R C) and ANSI C
- porting between HP C and Domain/C
- porting between HP C and VMS C
- calling other languages

General Portability Considerations

This section summarizes some of the general considerations to take into account when writing portable HP C programs. Some of the features listed here *may* be different on other implementations of C. Differences between Series 300/400 versus 700/800 implementations are also noted in this section.

Data Type Sizes and Alignments

Table 5-1 shows the sizes and alignments of the C data types on the different architectures. (On the 300/400, this applies to revision 5.15 and later.)

Table 5-1. C Data Types

Type	Size (in bytes)	Alignment (300/400)	Alignment (700/800)
`char`	1	byte	byte
`short`	2	2-byte	2-byte
`int`	4	4-byte[1]	4-byte
`long`	4	4-byte[1]	4-byte
`float`	4	4-byte[1]	4-byte
`double`	8	4-byte[1]	8-byte
`long double` (ANSI mode only)	16	4-byte[1]	8-byte
pointer	4	4-byte[1]	4-byte
`struct/union`		4-byte[1]	1-, 2-, 4- or 8-byte, depending on types of members
`enum`	4	4-byte (2-byte in a `struct`, array, or `union`)	4-byte (1-, 2-, or 4-byte if a `char`, `short`, or `int/long` type specifier is used during declaration)

1 Aligned on 2-byte boundary in `struct`, array, or `union`.

On Series 300/400, a structure ends on a 2-byte boundary; on Series 700/800, it ends on the same byte boundary as the start of the structure.

These default alignments can be overridden with data type alignment pragmas, described next.

Data Type Alignment Pragmas

Differences in data alignment can cause problems when porting code or data between systems that have different alignment schemes. For example, if you write a C program on Series 300/400 that writes records to a file, then read the file using the same program on Series 700/800, it may not work properly because the data may fall on different byte boundaries within the file due to alignment differences. To help alleviate this problem, HP C provides the `HP_ALIGN` pragma, which forces a particular alignment scheme, regardless of the architecture on which it is used. There are two forms of the `HP_ALIGN` pragma:

`#pragma HP_ALIGN` *align_scheme* [`PUSH`]

`#pragma HP_ALIGN POP`

align_scheme is one of the following:

`HPUX_WORD`	Use the Series 300/400 alignment scheme—the default alignment used on Series 300/400 systems.
`HPUX_NATURAL`	Use the Series 700/800 alignment scheme—the default alignment used on Series 700/800 systems.
`HPUX_NATURAL_S500`	Use the Series 500 alignment scheme—the default alignment used on Series 500 systems.
`NATURAL`	Use the natural alignment scheme—a mode created specifically for providing a consistent alignment scheme across all HP architectures.
`DOMAIN_WORD`	Use Domain word alignment mode from the Apollo Domain/C implementation.
`DOMAIN_NATURAL`	Use the Domain natural alignment mode from the Apollo Domain/C implementation.
`NOPADDING`	Causes all `struct` and `union` members that are not bit-fields to be packed as tightly as possible on a byte boundary.

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

The second pragma (#pragma HP_ALIGN POP) restores the alignment scheme that was saved at the last PUSH on the alignment scheme stack. If the alignment scheme stack is empty, the default alignment mode is used.

Note Using data type alignment pragmas can degrade performance. For details on using these pragmas, see the *HP C Programmer's Guide* for Series 300/400, or the *HP C/HP-UX Reference Manual* for Series 700/800.

Dereferencing Pointers to Unaligned Data

Prior to 9.0, to allow your Series 700/800 program to read and write the same binary files as your Series 300/400 program, HP_ALIGN pragmas could be used before and after the declaration of the structures that are written to disk. This could then be followed by compiling your program with the +u option. As a result, the compiler generated half-word access for *all* pointer dereferences which resulted in significant performance cost.

With 9.0, the compiler is now able to designate individual pointers to hold the addresses of non-natively aligned data enabling better code generation for properly aligned pointers. This is accomplished by using typedefs defined within the scope of the alignment pragma. Here is an example:

```
#pragma HP_ALIGN HPUX_WORD

struct t1 { char a; int b; } foo;

typedef int hw_aligned_int;

#pragma HP_ALIGN POP

main()
{
   int i;
   hp_aligned_int *p = &foo.b;

   i = *p;
}
```

In the presence of the `HP_ALIGN` pragma, the `typedef` will have alignment information carried with it.

As an alternative to using the above code, you can use the `allow_unaligned_data_access()` function, discussed below.

Accessing Unaligned Data

The Series 700/800 like all PA-RISC processors requires data to be accessed from locations that are aligned on multiples of the data size. The C compiler provides an option to access data from misaligned addresses using code sequences that load and store data in smaller pieces, but this option will increase code size and reduce performance. A bus error handling routine is also available to handle misaligned accesses but can reduce performance severely if used heavily.

Here are your specific alternatives for avoiding bus errors:

1. Change your code to eliminate misaligned data, if possible. This is the only way to get maximum performance, but it may be difficult or impossible to do. The more of this you can do, the less you'll need the next two alternatives.
2. Use the `+u`*number* compiler option available at 9.0 to allow 2-byte alignment. However, the `+u`*number* option, as noted above, creates big, slow code compared to the default code generation which is able to load a double precision number with one 8-byte load operation. Refer to the *HP C/HP-UX Reference Manual* (Series 700/800) for more information.
3. Finally, you can use `allow_unaligned_data_access()` to avoid alignment errors. `allow_unaligned_data_access()` sets up a signal handler for the SIGBUS signal. When the SIGBUS signal occurs, the signal handler extracts the unaligned data from memory byte by byte.

   To implement, just add a call to `allow_unaligned_data_access()` within your main program *before* the first access to unaligned data occurs. Then link with `-lhppa`. Any alignment bus errors that occur are trapped and emulated by a routine in the `libhppa.a` library in a manner that will be transparent to you. The performance degradation will be significant, but if it only occurs in a few places in your program it shouldn't be a big concern.

Whether you use alternative 2 or 3 above depends on your specific code.

The +u*number* option costs significantly less per access than the handler, but it costs you on every access, whether your data is aligned or not, and it can make your code quite a bit bigger. You should use it selectively if you can isolate the routines in your program that may be exposed to misaligned pointers.

There is a performance degradation associated with 3 because each unaligned access has to trap to a library routine. You can use the `unaligned_access_count` variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use 2. If you only occasionally use a misaligned pointer, it is probably better just use the `allow_unaligned_data_access` handler. There is a stiff penalty per bus error, but it doesn't cause your program to fail and it won't cost you anything when you operate on aligned data.

The following is a an example of its use within a C program:

```
extern int unaligned_access_count;
                  /* This variable keeps a count
                     of unaligned accesses. */


char arr[]="abcdefgh";
char *cp, *cp2;
int i=99, j=88, k;
int *ip;          /* This line would normally result in a
                     bus error on Series 700 or 800    */
main()
{
   allow_unaligned_data_access();
   cp = (char *)&i;
   cp2 = &arr[1];
   for (k=0; k<4; k++)
       cp2[k] = * (cp+k);
   ip = (int *)&arr[1];
   j = *ip;
   printf("%d\n", j);
   printf("unaligned_access_count is : %d\n", unaligned_access_count);
}
```

To compile and link this program, enter

```
cc filename.c -lhppa
```

This enables you to link the program with `allow_unaligned_data_access()` and the `int unaligned_access_count` that reside in `/usr/lib/libhppa.a`.

Note that there is a performance degradation associated with using this library since each unaligned access has to trap to a library routine. You can use the `unaligned_access_count` variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use the compiler option.

Checking for Alignment Problems with lint

If invoked with the `-s` option, the `lint` command generates warnings for C constructs that may cause portability and alignment problems between Series 300/400 and Series 700/800, and vice versa. Specifically, `lint` checks for these cases:

- Internal padding of structures. `lint` checks for cases where a structure member may be aligned on a boundary that is inappropriate according to the most-restrictive alignment rules. For example, given the code

  ```
  struct s1 { char c; long l; };
  ```

  `lint` issues the warning:

  ```
  warning: alignment of struct 's1' may not be portable
  ```

- Alignment of structures and simple types. For example, in the following code, the nested `struct` would align on a 2-byte boundary on Series 300/400 and an 8-byte boundary on Series 700/800:

  ```
  struct s3 { int i; struct { double d; } s; };
  ```

  In this case, `lint` issues this warning about alignment:

  ```
  warning: alignment of struct 's3' may not be portable
  ```

- End padding of structures. Structures are padded to the alignment of the most-restrictive member. For example, the following code would pad to a 2-byte boundary on Series 300/400 and a 4-byte boundary for Series 700/800:

  ```
  struct s2 { int i; short s; };
  ```

  In this case, `lint` issues the warning:

  ```
  warning:
  trailing padding of struct/union 's2' may not be portable
  ```

Note that these are only *potential* alignment problems. They would cause problems only when a program writes raw files which are read by another system. This is why the capability is accesible only through a command line option; it can be switched on and off.

`lint` does not check the layout of bit-fields.

Ensuring Alignment without Pragmas

Another solution to alignment differences between systems would be to define structures in such a way that they are forced into the same layout on different systems. To do this, use **padding** bytes—that is, dummy variables that are inserted solely for the purpose of forcing `struct` layout to be uniform across across implementations. For example, suppose you need a structure with the following definition:

```
struct S {
    char   c1;
    int    i;
    char   c2;
    double d;
};
```

An alternate definition of this structure that uses filler bytes to ensure the same layout on Series 300/400 and Series 700/800 would look like this:

```
struct S {
    char   c1;                  /* byte 0 */
    char   pad1,pad2,pad3;      /* bytes 1 through 3 */
    int    i;                   /* bytes 4 through 7 */
    char   c2;                  /* byte 8 */
    char   pad9,pad10,pad11,    /* bytes 9 */
           pad12,pad13,pad14,   /*  through */
           pad15;               /*    15 */
    double d;                   /* bytes 16 through 23 */
};
```

5

Casting Pointer Types

Before understanding how casting pointer types can cause portability problems, you must understand how Series 700/800 aligns data types. In general, a data type is aligned on a byte boundary equivalent to its size. For example, the `char` data type can fall on any byte boundary, the `int` data type must fall on a 4-byte boundary, and the `double` data type must fall on an 8-byte boundary. A valid *location* for a data type would then satisfy the following equation:

location mod `sizeof(`*data_type*`)` == 0

Consider the following program:

```
#include <string.h>
#include <stdio.h>
main()
{
  struct chStruct {
    char ch1;                   /* aligned on
                                   an even boundary */
    char chArray[9];            /* aligned on
                                   an odd byte boundary */
  } foo;

  int *bar;                     /* must be aligned
                                   on a word boundary */
    strcpy(foo.chArray, "1234");  /* place a value
                                     in the ch array */
    bar = (int *) foo.chArray;    /* type cast */
    printf("*bar = %d\n",*bar);   /* display the value */
  }
```

Casting a smaller type (such as **char**) to a larger type (such as **int**) will not cause a problem. However, casting a **char*** to an **int*** and then dereferencing the **int*** may cause an alignment fault. Thus, the above program crashes on the call to `printf()` when **bar** is dereferenced.

Such programming practices are inherently non-portable because there is no standard for how different architectures reference memory. You should try to avoid such programming practices.

As another example, if a program passes a casted pointer to a function that expects a parameter with stricter alignment, an alignment fault may occur. For example, the following program causes an alignment fault on Series 700/800:

```
void main (int argc, char *argv[])
{
    char   pad;
    char   name[8];

intfunc((int *)&name[1]);
}

int intfunc (int *iptr)
{
    printf("intfunc got passed %d\n", *iptr);
}
```

Type Incompatibilities and typedef

The C **typedef** keyword provides an easy way to write a program to be used on systems with different data type sizes. Simply define your own type equivalent to a provided type that has the size you wish to use.

For example, suppose system A implements **int** as 16 bits and **long** as 32 bits. System B implements **int** as 32 bits and **long** as 64 bits. You want to use 32 bit integers. Simply declare all your integers as type **INT32**, and insert the appropriate **typedef** on system A:

```
typedef long INT32;
```

The code on system B would be:

```
typedef int INT32;
```

Conditional Compilation

Using the **#ifdef** C preprocessor directive and the predefined symbols __hp9000s300, __hp9000s700, and __hp9000s800, you can group blocks of system-dependent code for conditional compilation, as shown below:

```
#ifdef  __hp9000s300
    :
  Series 300/400-specific code goes here...
    :
#endif

#ifdef  __hp9000s700
    :
  Series 700-specific code goes here...
    :
#endif

#ifdef  __hp9000s800
    :
  Series 700/800-specific code goes here...
    :
#endif
```

If this code is compiled on a Series 300/400 system, the first block is compiled; if compiled on Series 700, the second block is compiled; if compiled on *either* the Series 700 or the Series 800, the third block is compiled. You can use this feature to ensure that a program will compile properly on either Series 300/400 or 700/800.

If you want your code to compile *only* on the Series 800 but not on the 700, surround your code as follows:

```
#if (defined(__hp9000s800) && !defined(__hp9000s700))
    &vellipsis;
  Series 800-specific code goes here...
    &vellipsis;
#endif
```

Isolating System-Dependent Code with #include Files

`#include` files are useful for isolating the system-dependent code like the type definitions in the previous section. For instance, if your type definitions were in a file `mytypes.h`, to account for all the data size differences when porting from system A to system B, you would only have to change the contents of file `mytypes.h`. A useful set of type definitions is in `/usr/include/model.h`.

Note If you use the symbolic debugger, `xdb`, `include` files used within `union`, `struct`, or `array` initialization will generate correct code. However, such use is discouraged because `xdb` may show incorrect debugging information about line numbers and source file numbers.

Parameter Lists

On the Series 300/400, parameter lists grow towards higher addresses. On the Series 700/800, parameter lists are usually stacked towards decreasing addresses (though the stack itself grows towards higher addresses). The compiler may choose to pass some arguments through registers for efficiency; such parameters will have no stack location at all.

ANSI C function prototypes provide a way of having the compiler check parameter lists for consistency between a function declaration and a function call within a compilation unit. `lint` provides an option (`-Aa`) that flags cases where a function call is made in the absence of a prototype.

The ANSI C `<stdarg.h>` header file provides a portable method of writing functions that accept a variable number of arguments. Refer to the *HP C/HP-UX Reference Manual* or *stdarg*(5) for details and examples of the use of `<stdarg.h>`.

You should note that **<stdarg.h>** supersedes the use of the **varargs** macros. **varargs** is retained for compatibility with the pre-ANSI compilers and earlier releases of HP C/HP-UX. See *varargs*(5) and *vprintf*(3S) for details and examples of the use of **varargs**.

The char Data Type

The `char` data type defaults to signed. If a **char** is assigned to an **int**, sign extension takes place. A **char** may be declared **unsigned** to override this default. The line:

```
   unsigned char   ch;
```

declares one byte of unsigned storage named **ch**. On some non-HP-UX systems, `char` variables are unsigned by default.

Register Storage Class

The **register** storage class is supported on Series 300/400 and 700/800 HP-UX, and if properly used, can reduce execution time. Using this type should not hinder portability. However, its usefulness on systems will vary, since some ignore it. Refer to the *HP-UX Assembler and Supporting Tools* for Series 300/400 for a more complete description of the use of the **register** storage class on Series 300/400.

Also, the `register` storage class declarations are ignored when optimizing at levels 2 or greater on all Series.

Identifiers

To guarantee portable code to non-HP-UX systems, the ANSI C standard requires identifier names without external linkage to be significant to 31 case-sensitive characters. Names with external linkage (identifiers that are defined in another source file) will be significant to six case-insensitive characters. Typical C programming practice is to name variables with all lower-case letters, and **#define** constants with all upper case.

Predefined Symbols

The symbol `__hp9000s300` is predefined on Series 300/400; the symbols `__hp9000s800` and `__hppa` are predefined on Series 700/800; and `__hp9000s700` is predefined on Series 700 only. The symbols `__hpux` and `__unix` are predefined on all HP-UX implementations.

This is only an issue if you port code to or from systems that also have predefined these symbols.

Shift Operators

On left shifts, vacated positions are filled with 0. On right shifts of signed operands, vacated positions are filled with the sign bit (arithmetic shift). Right shifts of unsigned operands fill vacated bit positions with 0 (logical shift). Integer constants are treated as signed unless cast to unsigned. Circular shifts are not supported in any version of C. Shifts greater than 32 bits give an undefined result.

The sizeof Operator

The `sizeof` operator yields an `unsigned int` result, as specified in section 3.3.3.4 of the ANSI C standard (X3.159-1989). Therefore, expressions involving this operator are inherently unsigned. Do not expect any expression involving the `sizeof` operator to have a negative value (as may occur on some other systems). In particular, logical comparisons of such an expression against zero may not produce the object code you expect as the following example illustrates.

```
main()
{
       int i;
       i = 2;
 if ((i-sizeof(i)) < 0)          /* sizeof(i) is 4,
                                    but unsigned! */
  printf("test less than 0\n");
 else
  printf("an unsigned expression cannot be less than 0\n");
}
```

When run, this program will print

```
an unsigned expression cannot be less than 0
```

because the expression (i-sizeof(i)) is unsigned since one of its operands is unsigned (sizeof(i)). By definition, an unsigned number cannot be less than 0 so the compiler will generate an unconditional branch to the **else** clause rather than a test and branch.

Bit-Fields

The ANSI C definition does not prescribe bit-field implementation; therefore each vendor can implement bit-fields somewhat differently. This section describes how bit-fields are implemented in HP C.

Bit-fields are assigned from most-significant to least-significant bit on all HP-UX and Domain systems.

On all HP-UX implementations, bit-fields can be **signed** or **unsigned**, depending on how they are declared.

On the Series 300/400, a bit-field declared without the **signed** or **unsigned** keywords will be **signed** in ANSI mode and **unsigned** in compatibility mode by default.

On the Series 700/800, plain **int**, **char**, or **short** bit-fields declared without the **signed** or **unsigned** keywords will be **signed** in both compatibility mode and ANSI mode by default.

On the Series 700/800, and for the most part on the Series 300/400, bit-fields are aligned so that they cannot cross a boundary of the declared type. Consequently, some padding within the structure may be required. As an example,

```
struct foo
{
        unsigned int   a:3, b:3, c:3, d:3;
        unsigned int   remainder:20;
};
```

For the above struct, sizeof(struct foo) would return 4 (bytes) because none of the bit-fields straddle a 4 byte boundary. On the other hand, the following struct declaration will have a larger size:

```
struct foo2
{
        unsigned char   a:3, b:3, c:3, d:3;
        unsigned int    remainder:20;
};
```

In this struct declaration, the assignment of data space for c must be aligned so it doesn't violate a byte boundary, which is the normal alignment of unsigned char. Consequently, two undeclared bits of padding are added by the compiler so that c is aligned on a byte boundary. sizeof(struct foo2) returns 6 (bytes) on Series 300/400, and 8 on Series 700/800. Note, however, that on Domain systems or when using #pragma HP_ALIGN NATURAL, which uses Domain bit-field mapping, 4 is returned because the char bit-fields are considered to be ints.)

Bit-fields on HP-UX systems cannot exceed the size of the declared type in length. The largest possible bit-field is 32 bits. All scalar types are permissible to declare bit-fields, including enum.

Enum bit-fields are accepted on all HP-UX systems. On Series 300/400 in compatibility mode they are implemented internally as unsigned integers. On Series 700/800, however, they are implemented internally as signed integers so care should be taken to allow enough bits to store the sign plus the magnitude of the enumerated type. Otherwise your results may be unexpected. In ANSI mode, the type of enum bit-fields is signed int on *all* HP-UX systems.

Floating-Point Exceptions

In accordance with the IEEE standard, floating-point exceptions such as division by zero do *not* cause a trap using HP C on Series 700/800. By contrast, when using HP C on Series 300/400, floating-point exceptions will result in the run-time error message Floating exception (core dumped). One way to handle this error on Series 700/800 is by setting up a signal handler using the signal system call, and trapping the signal SIGFPE. For details, see *signal*(2), *signal*(5), and Chapter 12, "Advanced HP-UX Programming" in *Programming on HP-UX*.

For full treatment of floating-point exceptions and how to handle them, see *HP-UX Floating-Point Guide.*

Integer Overflow

In HP C, as in nearly every other implementation of C, integer overflow does not generate an error. The overflowed number is "rolled over" into whatever bit pattern the operation happens to produce.

Overflow During Conversion from Floating Point to Integral Type

HP-UX systems will report a `floating exception - core dumped` at run time if a floating point number is converted to an integral type and the value is outside the range of that integral type. As with the error described previously under "Floating-Point Exceptions", a program to trap the floating-point exception signal (`SIGFPE`) can be used. See *signal*(2) and *signal*(5) for details.

Structure Assignment

The HP-UX C compilers support structure assignment, structure-valued functions, and structure parameters. The structs in a `struct` assignment `s1=s2` must be declared to be the same `struct` type as in:

```
struct s  s1,s2;
```

Structure assignment is in the ANSI standard. Prior to the ANSI standard, it was a BSD extension that some other vendors may not have implemented.

Structure-Valued Functions

Structure-valued functions support storing the result in a structure:

```
s = fs();
```

All HP-UX implementations allow direct field dereferences of a structure-valued function. For example:

```
x = fs().a;
```

Structure-valued functions are ANSI standard. Prior to the ANSI standard, they were a BSD extension that some vendors may not have implemented.

Dereferencing Null Pointers

Dereferencing a null pointer has never been defined in any C standard. Kernighan and Ritchie's *The C Programming Language* and the ANSI C standard both warn against such programming practice. Nevertheless, some versions of C permit dereferencing null pointers.

Dereferencing a null pointer returns a zero value on all HP-UX systems. The Series 700/800 C compiler provides the `-z` compile line option, which causes the signal `SIGSEGV` to be generated if the program attempts to read location zero. Using this option, a program can "trap" such reads.

Since some programs written on other implementations of UNIX rely on being able to dereference null pointers, you may have to change code to check for a null pointer. For example, change:

```
if (*ch_ptr != '\0')
```

to:

```
if ((ch_ptr != NULL) && *ch_ptr != '\0')
```

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 acts as if the `-z` flag is always set.

Expression Evaluation

The order of evaluation for some expressions will differ between HP-UX implementations. This does not mean that operator precedence is different. For instance, in the expression:

```
x1 = f(x) + g(x) * 5;
```

`f` may be evaluated before or after `g`, but `g(x)` will always be multiplied by `5` before it is added to `f(x)`. Since there is no C standard for order of evaluation of expressions, you should avoid relying on the order of evaluation when using functions with side effects or using function calls as actual parameters. You should use temporary variables if your program relies upon a certain order of evaluation.

Variable Initialization

On some C implementations, `auto` (non-`static`) variables are implicitly initialized to 0. This is *not* the case on HP-UX and it is most likely not the case on other implementations of UNIX. *Don't depend on the system initializing your local variables*; it is not good programming practice in general and it makes for nonportable code.

Conversions between unsigned char or unsigned short and int

5

All HP-UX C implementations, when used in compatibility mode, are **unsigned preserving**. That is, in conversions of `unsigned char` or `unsigned short` to `int`, the conversion process first converts the number to an `unsigned int`. This contrasts to some C implementations that are **value preserving** (that is, `unsigned char` terms are first converted to `char` and then to `int` before they are used in an expression).

Consider the following program:

```
main()
{
 int i = -1;
 unsigned char uc = 2;
 unsigned int ui = 2;

 if (uc > i)
  printf("Value preserving\n");
 else
  printf("Unsigned preserving\n");
 if (ui < i)
  printf("Unsigned comparisons performed\n");
}
```

On HP-UX systems in compatibility mode, the program will print:

```
Unsigned preserving
Unsigned comparisons performed
```

In contrast, ANSI C specifies value preserving; so in ANSI mode, all HP-UX C compilers are value preserving. The same program, when compiled in ANSI mode, will print:

```
Value preserving
Unsigned comparisons performed
```

This is covered in more detail in the section "Silent Changes for ANSI C", later in this chapter.

Temporary Files ($TMPDIR)

All HP-UX C compilers produce a number of intermediate temporary files for their private use during the compilation process. These files are normally invisible to you since they are created and removed automatically. If, however, your system is tightly constrained for file space these files, which are usually generated on `/tmp` or `/usr/tmp`, may exceed space requirements. By assigning another directory to the `TMPDIR` environment variable you can redirect these temporary files. See the `cc` manual page for details.

5

Compile Line Options

There are some minor differences in HP-UX C compiler options. You may have to modify makefiles if they use any of the options listed in Table 5-2. Be aware that the purpose of the table below is only to point out differences between implementations. Therefore, you should see *cc*(1) for more details on using these options, or refer to the *HP C/HP-UX Reference Manual* (for Series 700/800 information) and the *C Programmer's Guide* (for Series 300/400 information).

Table 5-2. Differences in C Compile Line Options

Option	Effect	Difference
`+a`	Do not assemble with prefix file.	Series 700/800 only.
`+bfpa`	Floating-point option.	Series 300/400 only.
`+DA1.0`	Optimize for Series 800 architecture and instruction set. Or, use `DA8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only.
`+DA1.1`	Optimize for Series 700 architecture and instruction set. Or, use `DA7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only.
`+df`*name*	Specifies the profile database to use with profile-based optimization and the `+P` command line option.	Series 700/800 only.
`+DS1.0`	Optimize for Series 800 instruction scheduling. Or, use `DS8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only.
`+DS1.1`	Optimize for Series 700 instruction scheduling. Or, use `DS7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only.
`+e`	Enables extensions and an `HPUX_SOURCE` name space when compiling in ANSI C mode.	System-dependent options.
`+f`	Same as `+r` except promotions do not occur for parameters and values returned from functions.	Series 700/800 only.
`+ESlit`	Places string literals and constants into the `$LIT` subspace.	Series 700/800 only.
`+ESsfc`	Replaces millicode calls with inline code when performing simple function pointer comparisons.	Series 700/800 only.
`+ffpa`	Floating-point option.	Series 300/400 only.

Table 5-2. Differences in C Compile Line Options (continued)

Option	Effect	Difference
`+FP`*flags*	Initializes the flags that specify how floating-point exceptions should be trapped.	Series 700/800 only.
`+I`	Instructs the compiler to prepare object code for profiling.	Series 700/800 only.
`+L`	Enable listing facility and listing pragmas.	Series 700/800 only.
`+Lp`	Same as `+L` but includes post-processed source file.	Series 700/800 only.
`+M`	Floating point option.	Series 300/400 only.
`+m`	Cause identifier maps to be printed.	Series 700/800 only.
`+N`	Adjusts size of internal compiler tables.	Series 300/400 only.
`-N`	Create non-shareable executeable.	Such executables cannot be executed by **`exec`** on Series 700/800.
`+O`*opt*	Specify optimization level.	Semantics differ.
`+Obb`*num*	Specify maximum basic blocks allowed in procedure optimized at level 2.	Series 700/800 only.
`+o`	Print code offsets in hexadecimal at end of listing.	Series 700/800 only.
`+P`	Directs the compiler to use profile information to guide code generation and profile-based optimization.	Series 700/800 only.
`+pgm`*name*	Used with profile-based optimization and the `+P` command line option.	Series 700/800 only.
`+R`*n*	Allow only the first *n* **`register`** variables to actually have **`register`** storage class.	Series 700/800 only.
`+r`	Inhibit automatic promotion to **`float`** or **`double`** when evaluating expressions and passing arguments in compatibility mode.	Series 700/800 only.

Table 5-2. Differences in C Compile Line Options (continued)

Option	Effect	Difference
`+u`*number*	Pointers use 2-byte addressing.	Series 700/800 only.
`-W`	Pass options to subprocesses.	System-dependent options.
`+w`*n*	Specify level of warning messages.	Series 700/800 only.
`+`*opt*	Shorthand for some `-W` options.	System-dependent options.
`-Z`	Allow dereferencing of null pointers.	Not supported on Series 300/400. Is the default on Series 700/800.
`-z`	Allow run-time detection of null pointers.	Series 700/800 only.

Input/Output

Since the C language definition provides no I/O capability, it depends on library routines supplied by the host system. Data files produced by using the HP-UX calls *write*(2) or *fwrite*(3) should not be expected to be portable between different system implementations. Byte ordering and structure packing rules will make the bits in the file system-dependent, even though identical routines are used. When in doubt, move data files using ASCII representations (as from *printf*(3)), or write translation utilities that deal with the byte ordering and alignment differences.

Checking for Standards Compliance

As discussed in Chapter 2, writing programs that comply with industry standards helps to ensure that your code will be portable.

In order to check for standards compliance to a particular standard, you can use the `lint` program with one of the following `-D` options:

- `-D_XOPEN_SOURCE`
- `-D_POSIX_SOURCE`

For example, the command

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

checks the source file `file.c` for compliance with the POSIX standard.

If you have the HP Advise product, you can also check for C standard compliance using the `apex` command.

Porting between K&R C and ANSI C

This section describes porting C programs compliant with the Kernighan & Ritchie language definition to ANSI C compliance. Specifically, it discusses:

- Compile line options.
- ANSI C name spaces.
- Differences that may cause porting problems.

Compile Line Options

By default, HP C compilers use **compatibility mode**; that is, HP C compilers use the language definition from Kernighan & Ritchie's *The C Programming Language*, First Edition, as well as selected BSD extensions. To compile using ANSI C mode, specify the `-Aa` compile line option. (`-Ac` is the compile line option for compiling in compatibility mode.)

The `lint` program can be used to find non-standard language features within a program. It can also produce huge amounts of other information as well. Once you find non-standard features, you can then go into the source program and fix them.

On Series 700/800, you can also specify the `+w1` option for `cc`. This option is especially useful in that it generates warning messages only for the use of non-standard features, unlike `lint`, which generates messages for other things as well.

How 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 without 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 an *open* system call on any other operating system.

HP Header File and Library Implementation of Name Space

The HP header files and libraries have been designed to support four different name spaces, as shown in Figure 5-1.

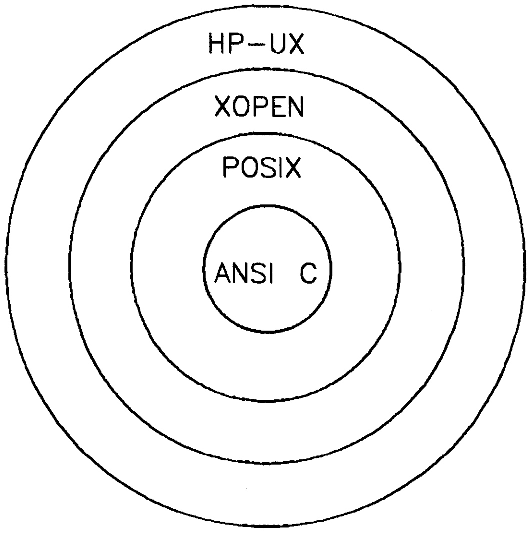

Figure 5-1. ANSI C Name Spaces

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. These names are a superset of XOPEN.

The HP library implementation has been designed with the assumption that many existing programs will use more routines than those specified 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. Table 5-3 provides information on how to select a name space from a command line or from within a program using the defined libraries in ANSI mode.

Table 5-3. 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 on Series 300/400 because `socket.h` uses the symbol `u_short` which 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;
         ^
```

On Series 700/800, the following error message is produced:

```
"/usr/include/sys/socket.h", line 79: error 1000:
   Unexpected symbol: "u_short".
```

This error may be fixed by adding `-D_HPUX_SOURCE` to the command line when compiling.

Silent Changes for ANSI C

This section describes the situations that occur when ANSI C mode silently has different behavior from compatibility mode. Many of these silent behaviors can be detected by running `lint`, as described at the start of this section. The following list provides some of these silent behaviors.

Note The following list does not document *all* differences between HP C and ANSI C. For a more detailed description of differences between K&R C and ANSI C, refer to one of the following books:

- *A Book on C*, 2nd ed., Kelley and Pohl, Benjamin/Cummings.
- *The C Programming Language*, 2nd ed., Kernighan and Ritchie, Prentice Hall.

- On Series 300/400, a bit-field declared without the `signed` or `unsigned` keywords will be `signed` in ANSI mode and `unsigned` in compatibility mode. On Series 700/800, bit-fields are `signed` in both compatibility and ANSI 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 on Series 300/400, read *C: A Reference Manual*. On Series 700/800, refer to the *HP C/HP-UX Reference Manual*.

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

```
main(){
  unsigned short us = 1;
  int i = -2;
  if (i > us)
    printf("compatibility mode\n"); /* promoted to unsigned int */
  else
    printf("ANSI mode\n");          /* promoted to int */
}
```

5

- In general, promotion rules for resultant expression types have changed.

  Figure 5-2 shows promotion rules under compatibility mode. The resultant type is the lowest common parent in the tree for the operands. For example, if two operands in an expression are of type **char** and **short**, the resultant expression type is **int**; if the expression contains three operands of type **short**, **int**, and **unsigned char**, the expression type is unsigned int".

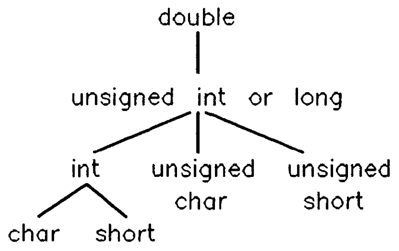

Figure 5-2. Compatibility Mode Promotion Rules

Figure 5-3 shows the promotion rules under ANSI mode. For example, an expression involving long and **unsigned int** operands results in an **unsigned** long result.

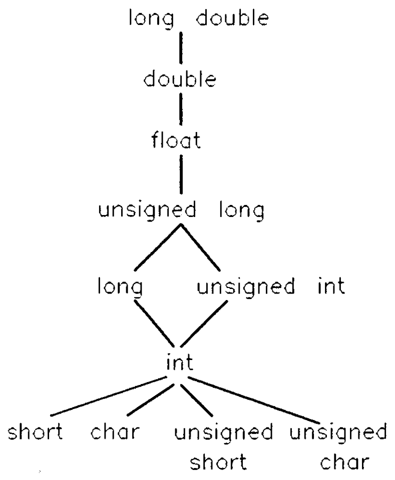

Figure 5-3. ANSI Mode Promotion Rules

In expressions involving shift operators (`<<` and `>>`), the resulting expression has the same type as the promoted left operand.

- Floating-point expressions with `float` operands will automatically be computed as `float` precision in ANSI mode.

  In compatibility mode, `floats` are always computed in `double` precision. On the Series 700/800, using the `+f` option will keep `floats` as `floats` rather than promoting them to `doubles`. Or, `+r` can be used.

  Keeping `floats` as `floats` using ANSI mode or the above options in compatibility mode) will result in faster running programs.

- Initialization rules are different in some cases when braces are omitted in an initialization of an aggregate or union object.

- Unsuffixed integer constants may have different types. In compatibility mode, unsuffixed constants have type **int**. In 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 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 ANSI mode. The term block scopes refers to identifiers declared inside a block or list of parameter declarations in a function definition, having meaning from their point of declaration to the end of the block. In 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 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.*

5

- On Series 700/800, variable shifts (<< or >>) where the right operand has a value greater than 31 or less than 0 will no longer always have a result of 0. For example,

  ```
  unsigned int i,j = 0xffffffff, k = 32;
  i = j >> k;  /* i gets the value 0 in compatibility mode,  */
               /* 0xffffffff(-1) in ANSI mode.  */
  ```

Porting between HP C and Domain/C

All HP-UX and Domain computers have ANSI C compilers. Strictly standard-compliant programs are highly portable between all these architectures.

The following Domain/C extensions are *not* supported on HP-UX in compatibility mode and in most cases, are *not* supported in ANSI mode either:

- Reference variables.
- The following preprocessor directives: `#attribute`, `#options`, `#section`, `#module`, `#debug`, `#eject`, `#list`, `#nolist`, and `#systype`.
- `std_$call`.
- `__attribute` modifier and `__options` specifier.
- `systype` predefined macro.
- `_BFMT__COFF` predefined macro.
- `_ISP__M68K` predefined macro.
- `_ISP__A88K` predefined macro.
- `_ISP__PA_RISC` predefined macro.
- Partial specification of `struct` and `union` members.

Function prototypes, `struct` and `union` initialization, and the predefined names `__DATE__` and `__TIME__`, all of which are ANSI C features, are supported on HP-UX in ANSI mode.

Compile line options are different between HP-UX C and Domain/C. Check the respective *cc*(1) page for complete descriptions.

There are other differences between HP-UX C and Domain/C:

- Alignment: All Domain workstations have hardware or software assists to handle misaligned data. Programs that rely on these features will not run on the Series 800.
- Floating-point exceptions: All Domain workstations, by default, enable invalid operation, divide by zero, and overflow exception traps. Programs that rely on fault detection, for instance, to enter a fault handler or to terminate execution on encountering a fault, will ordinarily generate useless

output on HP-UX. However, the PA1.1 math library for the Series 700/800 provides a function `fpsetdefaults`(3M), which enables these traps and therefore allows such programs to run as expected. For more information, see the *HP-UX Floating-Point Guide*.

- `struct` layout and alignment, especially bit-field, is different.
- `float` data type: Domain/C optimizes a statement all of whose atoms are `float` or floating-point constants, to be evaluated in `float` rather than `double`.
- `register` declarations: Domain/C completely ignores `register` declarations, except to ensure that language constraints are not violated.
- Include file search rules are different.
- Programs that rely on undefined behaviors, for instance, the order of expression evaluation and the application of unsequenced side-effects, will probably execute differently.

Porting between HP C and VMS C

The C language itself is easy to port from VMS to HP-UX for two main reasons:

- There is a high degree of compatibility between HP C and other common industry implementations of C as well as within the HP-UX family.
- The C language itself does not consider file manipulation or input/output to be part of the core language. These issues are handled via libraries. Thus, C avoids some of the thorniest issues of portability.

In most cases, HP C (in compatibility mode) is a superset of VMS C. Therefore, porting from VMS to HP-UX is easier than porting in the other direction. The next several subsections describe features of C that can cause problems in porting.

Core Language Features

- Basic data types in VMS have the same general sizes as their counterparts on HP-UX. In particular, all integral and floating- point types have the same number of bits. `structs` and `unions` do not necessarily have the same size because of different alignment rules.
- Basic data types are aligned on arbitrary byte boundaries in VMS C. HP-UX counterparts generally have more restrictive alignments, as shown in Table 5-1.
- Type `char` is signed by default on both VMS and HP-UX.
- The `unsigned` adjective is recognized by both systems and is usable on `char`, `short`, `int`, and `long`. It can also be used alone to refer to `unsigned int`.
- Both VMS and HP-UX support `void` and `enum` data types although the allowable uses of `enum` vary between the two systems. HP-UX is generally less restrictive.
- The VMS C storage class specifiers `globaldef`, `globalref`, and `globalvalue` have no direct counterparts on HP-UX or other implementations of UNIX. On HP-UX, variables are either local or global, based strictly on scope or `static` class specifiers.

- The VMS C class modifiers **readonly** and **noshare** have no direct counterparts on HP-UX.
- **structs** are packed differently on the two systems. All elements are byte aligned in VMS whereas they are aligned more restrictively on the different HP-UX architectures based upon their type. Organization of fields within the **struct** differs as well.
- Bit-fields within **structs** are more general on HP-UX than on VMS. VMS requires that they be of type **int** or **unsigned** whereas they may be any integral type on HP-UX.
- Assignment of one **struct** to another is supported on both systems. However, VMS permits assignment of **structs** provided the types of both sides have the same size. HP-UX is more restrictive because it requires that the two sides be of the same type.

- VMS C stores floating-point data in memory using a proprietary scheme. Floats are stored in **F_floating** format. Doubles are stored either in **D_floating** format or **G_floating** format. **D_floating** format is the default. HP-UX uses IEEE standard formats which are not compatible with VMS types but which are compatible with most other industry implementations of UNIX.
- VMS C converts floats to doubles by padding the mantissa with 0s. HP-UX uses IEEE formats for floating-point data and therefore must do a conversion by means of floating-point hardware or by use of library functions. When doubles are converted to floats in VMS C, the mantissa is rounded toward zero, then truncated. HP-UX uses either floating point hardware or library calls for these conversions.

  The VMS **D_floating** format can hide programming errors. In particular, you might not immediately notice that mismatches exist between formal and actual function arguments if one is declared **float** and the counterpart is declared **double** because the only difference in the internal representation is the length of the mantissa.

- Due to the different internal representations of floating-point data, the range and precision of floating-point numbers differs on the two systems according to the following tables:

Table 5-4. VMS C Floating-Point Types

Format	Approximate Range of $\|x\|$	Approximate Precision
`F_floating`	0.29E-38 to 1.7E38	7 decimal digits
`D_floating`	0.29E-38 to 1.7E38	16 decimal digits
`G_floating`	0.56E-308 to 0.99E308	15 decimal digits

Table 5-5. HP-UX C Floating-Point Types

Format	Approximate Range of $\|x\|$	Approximate Precision
`float`	1.17E-38 to 3.40E38	7 decimal digits
`double`	2.2E-308 to 1.8E308	16 decimal digits
`long double`	3.36E-4932 to 1.19E4932	31 decimal digits

- VMS C identifiers are significant to the 31st character. HP-UX C identifiers are significant to 255 characters.
- `register` declarations are handled differently in VMS. The `register` reserved word is regarded by the compiler to be a strong hint to assign a dedicated register for the variable. On Series 300/400, the `register` declaration causes an integral or pointer type to be assigned a dedicated register to the limits of the system, unless optimization at level `+O2` or greater is requested, in which case the compiler ignores `register` declarations. Series 700/800 treats `register` declarations as hints to the compiler.
- If a variable is declared to be `register` in VMS and the `&` address operator is used in conjunction with that variable, no error is reported. Instead, the VMS compiler converts the class of that variable to `auto`. HP-UX compilers will report an error.

- Type conversions on both systems follow the usual progression found on implementations of UNIX.
- Character constants (not to be confused with string constants) are different on VMS. Each character constant can contain up to four ASCII characters. If it contains fewer, as is the normal case, it is padded on the left by NULLs. However, only the low order byte is printed when the `%c` descriptor is used with `printf`. Multicharacter character constants are treated as an overflow condition on Series 300/400 if the numerical value exceeds 127 (the overflow is silent). In compatibility mode, Series 700/800 detects all multicharacter character constants as error conditions and reports them at compile time.
- String constants can have a maximum length of 65535 characters in VMS. They are essentially unlimited on HP-UX.
- VMS provides an alternative means of identifying a function as being the main program by the use of the adjective **main program** that is placed on the function definition. This extension is not supported on HP-UX. Both systems support the special meaning of `main()`, however.
- VMS implicity initializes pointers to 0. HP-UX makes no implicit initialization of pointers unless they are `static`, so dereferencing an uninitialized pointer is an undefined operation on HP-UX.
- VMS permits combining type specifiers with `typedef` names. So, for example:

  ```
  typedef long t;
  unsigned t x;
  ```

  is permitted on VMS. This is permitted only in compatibility mode on Series 300/400; it is not allowed in ANSI C mode on any HP-UX system. To accomplish this on Series 700/800, change the `typedef` to include the type specifier:

  ```
  typedef unsigned long t;
  t x;
  ```

  Or use a `#define`:

  ```
  #define t long
  unsigned t x;
  ```

Preprocessor Features

- VMS supports an unlimited nesting of `#includes`. HP-UX in compatibility mode guarantees 35 levels of nesting. HP-UX in ANSI mode guarantees 57 levels of nesting.
- The algorithms for searching for `#includes` differs on the two systems. VMS has two variables, `VAXC$INCLUDE` and `C$INCLUDE` which control the order of searching. HP-UX follows the usual order of searching found on most implementations of UNIX.
- `#dictionary` and `#module` are recognized in VMS but not on HP-UX.
- The following symbols are predefined in VMS but not on HP-UX: `vms`, `vax`, `vaxc`, `vax11c`, `vms_version`, `CC$gfloat`, `VMS`, `VAX`, `VAXC`, `VAX11C`, and `VMS_VERSION`.
- The following symbols are predefined on all HP-UX systems but not in VMS:

  `__hp9000s300` on Series 300/400
  `__hp9000s700` on Series 700
  `__hp9000s800` on Series 700/800
  `__hppa` on Series 700/800
  `__hpux` and `__unix` on all systems

- HP-UX preprocessors do not include white space in the replacement text of a macro. The VMS preprocessor does include the trailing white space. If your HP C program depends on the inclusion of the white space, you can place white space around the macro invocation.

Compiler Environment

- In VMS, files with a suffix of `.C` are assumed to be C source files, `.OBJ` suffixes imply object files, and `.EXE` suffixes imply executable files. HP-UX uses the normal conventions on UNIX that `.c` implies a C source file, `.o` implies an object file, and `a.out` is the default executable file (but there is no other convention for executable files).
- `varargs` is supported on VMS and all HP-UX implementations. See *vprintf*(3S) and *varargs*(5) for a description and examples.
- `curses` is supported on VMS and all HP-UX implementations. See *curses*(3X) for a description.

- VMS supports **VAXC$ERRNO** and **errno** as two system variables to return error conditions. HP-UX supports **errno** although there may be differences in the error codes or conditions.
- VMS supplies **getchar** and **putchar** as functions only, not as macros. HP-UX supplies them as macros and also supplies the functions **fgetc** and **fputc** which are the function versions.
- Major differences exist between the file systems of the two operating systems. One of these is that the VMS directory **SYS$LIBRARY** contains many standard definition files for macros. The HP-UX directory **/usr/include** has a rough correspondence but the contents differ greatly.
- A VMS user must explicitly link the RTL libraries SYS$LIBRARY:VAXCURSE.OLB, SYS$LIBRARY:VAXCRTLG.OLB or SYS$LIBRARY:VAXCRTL.OLB to perform C input/output operations. The HP-UX input/output utilities are included in **/lib/libc**, which is linked automatically by cc without being specified by the user.
- Certain standard functions may have different interfaces on the two systems. For example, strcpy() copies one string to another but the resulting destination may not be NULL terminated on VMS whereas it always will be on HP-UX.
- The commonly used HP-UX names **end**, **edata** and **etext** are not available on VMS.

Calling Other Languages

It is possible to call a routine written in another language from a C program, but you should have a good reason for doing so. Using more than one language in a program that you plan to port to another system will complicate the process. In any case, make sure that the program is thoroughly tested in any new environment.

If you do call another language from C, you will have the other language's anomalies to consider plus possible differences in parameter passing. Since all HP-UX system routines are C programs, calling programs written in other languages should be an uncommon event. If you choose to do so, remember that C passes all parameters by value except arrays and structures. The ramifications of this depend on the language of the called function, as shown in Table 5-6.

Table 5-6. C Interfacing Compatibility

C	HP-UX Pascal	FORTRAN
char	none	byte
unsigned char	char	character (could reside on an odd boundary and cause a memory fault)
char * (string)	none	none
unsigned char * (string)	PAC+chr(0) (PAC = packed array[1..*n*] of char)	Array of char+char(0)
short (int)	-32768..32767 (shortint on Series 700/800)	integer*2
unsigned short (int)	BIT16 on Series 700/800; none on Series 300/400 (0..65535 will generate a 16-bit value only if in a packed structure)	none
int	integer	integer (*4)
long (int)	integer	integer (*4)
unsigned (int)	none	none
float	real	real (*4)
double	longreal	real*8
long double[1]	none	real*16
type* (pointer)	^var, pass by reference, or use anyvar	none
&var (address)	addr(var) (requires $SYSPROG$)	none
*var (deref)	var^	none
struct	record (cannot always be done; C and Pascal use different packing algorithms)	structure
union	record case of ...	union

1 long double is available only in ANSI mode.

5

Calling FORTRAN

You can compile FORTRAN functions separately by putting the functions you want into a file and compiling it with the `-c` option to produce a `.o` file. Then, include the name of this `.o` file on the `cc` command line that compiles your C program. The C program can refer to the FORTRAN functions by the names they are declared by in the FORTRAN source.

Remember that in FORTRAN, parameters are usually passed by reference (except `CHARACTER` parameters on Series 700/800, which are passed by descriptor), so actual parameters in a call from C must be pointers or variable names preceded by the address-of operator (&).

The following program uses a FORTRAN block data subprogram to initialize a common area and a FORTRAN function to access that area:

```
      double precision function get_element(i,j)
      double precision array
      common /a/array(1000,10)
      get_element = array(i,j)
      end

      block data one
      double precision array
      common /a/array(1000,10)
C Note how easily large array initialization is done.
      data array /1000*1.0,1000*2.0,1000*3.0,1000*4.0,1000*5.0,
     *  1000*6.0,1000*7.0,1000*8.0,1000*9.0,1000*10.0/
      end
```

The FORTRAN function and block data subprogram contained in file **`xx.f`** are compiled using `f77 -c xx.f`.

The C main program is contained in file **`x.c`**:

```
main()
{
int i;

extern double get_element(int *, int *);

  for (i=1; i <= 10; i++)
      printf("element = %f\n", get_element(&i,&i));
}
```

The C main program is compiled using `cc -Aa x.c xx.o`.

5

Another area for potential problems is passing arrays to FORTRAN subprograms. An important difference between FORTRAN and C is that FORTRAN stores arrays in column-major order whereas C stores them in row-major order (like Pascal).

For example, the following shows sample C code:

```
int i,j;
int array[10][20];

for (i=0; i<10; i++) {
    for (j=0; j<20; j++)    /* Here the 2nd dimension
                               varies most rapidly */
    array [i][j]=0;
}
```

Here is similar code for FORTRAN:

```
integer array (10,20)

do J=1,20
  do I=1,10        !Here the first dimension varies most rapidly
    array(I,J)=0
  end do
end do
```

Therefore, when passing arrays from FORTRAN to C, a C procedure should vary the first array index the fastest. This is shown in the following example in which a FORTRAN program calls a C procedure:

```
integer array (10,20)

do j=1,20
  do i=1,10
    array(i,j)=0
  end do
end do
call cproc (array)
     .
     .
     .
cproc (array)
int array [][];

for (j=1; j<20; j++) {
    for (i=1; i<20; i++)   /* Note that this is the reverse from
                             how you would normally access the
                             array in C as shown above */
    array [i][j]= ...
}
         .
         .
         .
```

There are other considerations as well when passing arrays to FORTRAN subprograms. For details, see *HP-UX Assembler Reference and Supporting Documents* (for Series 300/400), or *HP C Programmer's Guide* (for Series 700/800).

It should be noted that a FORTRAN `main` should *not* be linked with `cc`.

Calling Pascal

Pascal gives you the choice of passing parameters *by value* or *by reference* (var parameters). C passes all parameters (other than arrays and structures) by value, but allows passing pointers to simulate pass by reference. If the Pascal function does not use var parameters, then you may pass values just as you would to a C function. Actual parameters in the call from the C program corresponding to formal var parameters in the definition of the Pascal function should be pointers.

Arrays correlate fairly well between C and Pascal because elements of a multidimensional array are stored in row-major order in both languages. That is, elements are stored by rows; the rightmost subscript varies fastest as elements are accessed in storage order.

Note that C has no special type for boolean or logical expressions. Instead, any integer can be used with a zero value representing false, and non-zero representing true. Also, C performs all integer math in full precision (32-bit); the result is then truncated to the appropriate destination size.

The basic method for calling Pascal functions on the Series 300/400 is to put the Pascal function into a module that exports the function, compile that file using `pc -c`, and then link it with your main C program by including the name of the Pascal `.o` file on the `cc` command line.

To call Pascal procedures from C on the Series 700/800, a program may first have to call the Pascal procedure U_INIT_TRAPS. See the *HP Pascal Programmer's Guide* for details about the TRY/RECOVER mechanism.

As true of FORTRAN mains, a Pascal main should *not* be linked with cc.

To call Pascal procedures from C or FORTRAN on the Series 300/400, the user must first call the procedure asm_initproc to initialize the heap, initialize the escape (TRY/RECOVER) mechanism, and set up the standard files input, output, and stderr. At the end, a call to asm_wrapup should be made. To work correctly, asm_initproc must be called with the value 0 or 1 (0 = buffered input; 1 = unbuffered input) as a parameter by reference (i.e., a pointer to 0). Without this parameter, asm_initproc generates a memory fault. An example is shown below.

The Series 300/400 C program shown below calls two Pascal integer functions:

```
main() /* The C main program */
{

 int noe = 1;
 int *c, *a_cfunc(), *a_dfunc();
 int *noecho = &noe;

 asm_initproc(noecho); /* Pascal initialization */
 c = a_cfunc();
 printf("%d\n",c);
 c = a_dfunc();
 printf("%d\n",c);
 asm_wrapup();  /* Pascal closure */
}
```

The following source is the Pascal module:

```
  module a;
export
 function cfunc : integer;
 function dfunc : integer;

implement
 function cfunc : integer;
  var x : integer;

  begin
   x := MAXINT;
   cfunc := x;
  end;

 function dfunc : integer;
  var x : integer;

  begin
   x := MININT;
   dfunc := x;
  end;
  end.
```

5

The command line for producing the Pascal relocatable object is

```
$ pc -c pfunc.p
```

On Series 300/400, the command line for compiling the C main program and linking the Pascal module is

```
$ cc x.c pfunc.o -lpc
```

or, on Series 700/800, it is

```
$ cc x.c pfunc.o -lcl
```

The following output results:

```
2147483647
```

on Series 300/400 and

```
2147483647
-2147483648
```

on Series 700/800.

6 Porting FORTRAN Programs

Prior to the HP-UX 9.0 release, there were many differences between the FORTRAN compilers on the Series 300/400/700 and 800. With the HP-UX 9.0 release, the compilers have been merged. As a result, the only remaining differences are due to the differences between the Motorola-based and PA-RISC-based architectures. You may experience some problems recompiling Series 800 programs using 9.0 due to minor differences.

The HP-UX FORTRAN compiler implements the full ANSI FORTRAN 77 language and MIL-STD-1753 standard as well as HP's extensions. In addition, many common extensions found in other non-HP-UX implementations have been added, particularly those from FORTRAN 7x on HP 1000 systems and VAX™ VMS FORTRAN. This chapter describes:

- general considerations for writing portable FORTRAN programs
- porting between Series 300/400 and Series 700/800
- porting FORTRAN programs between HP-UX and VMS
- calling other languages

Note Refer to the *FORTRAN/9000 Programmer's Guide* for additional portability information.

General Portability Considerations

Although FORTRAN on HP 9000 computers follows all the relevant standards, there are some extended features that may not port easily from one system to another. This section summarizes general considerations you should be aware of when porting FORTRAN programs to and from HP-UX systems across various other vendors implementations. Some of the information in this section will also be useful for porting FORTRAN programs between Series 300/400 and Series 700/800, the entire focus of the next section.

Data Type Sizes and Alignment

Table 6-1 shows the sizes and alignments of the FORTRAN data types on Series 300/400 and 700/800 computers.

Table 6-1. FORTRAN Data Types

Type	Size (bytes)	Alignment (300/400)[1]	Alignment (700/800)[1]
CHARACTER	1	1-byte	1-byte
Hollerith[2]	1	1-byte	1-byte
BYTE,LOGICAL*1[2,3]	1	1-byte	1-byte
LOGICAL*2[2,3]	2	2-byte	2-byte
INTEGER*2[2,3]	2	2-byte	2-byte
LOGICAL*4[3]	4	4-byte (2-byte within record)	4-byte
INTEGER, INTEGER*4[3]	4	4-byte (2-byte within record)	4-byte
REAL, REAL*4[3]	4	4-byte (2-byte within record)	4-byte
REAL*16[2,3]	16	4-byte (2-byte within record)	8-byte
DOUBLE PRECISION,REAL*8[3]	8	4-byte (2-byte within record)	8-byte
COMPLEX*8[3]	8	4-byte (2-byte within record)	8-byte
DOUBLE COMPLEX, COMPLEX*16[2,3]	16	4-byte (2-byte within record)	8-byte
RECORD		4-byte (2-byte within array or another record; alignment alterable using $NOSTANDARD ALIGNMENT)	Aligned on most restrictive field.

1 2-byte alignment for items 4 bytes and larger when **$HP1000 ALIGNMENT ON** is specified on all Series. Note that this alignment causes slower execution due to use of halfword load/store instructions.

2 This type is an extension to the ANSI FORTRAN 77 standard.

3 ANSI does not support a length descriptor *n.

If the +A compile line option is specified, then any non-character data types larger than two bytes are aligned on a 2-byte boundary instead of on 4-byte boundaries, for all HP 9000 implementations. This allows you to align data using alignment rules from previous releases of FORTRAN. The $HP1000 ALIGNMENT ON directive performs the same function as the +A option.

Accessing Unaligned Data

The Series 700/800 like all PA-RISC processors requires data to be accessed from locations that are aligned on multiples of the data size. The FORTRAN compiler provides an option to access data from misaligned addresses using code sequences that load and store data in smaller pieces, but this option will increase code size and reduce performance. A bus error handling routine is also available to handle misaligned accesses but can reduce performance severely if used heavily.

Here are your specific alternatives for avoiding bus errors:

1. Change your code to eliminate misaligned data, if possible. This is the only way to get maximum performance, but it may be difficult or impossible to do. The more of this you can do, the less you'll need the next two alternatives.

2. To allow 2-byte alignment of 4- and 8-byte items in FORTRAN, use `+A`. It allows 4- and 8-byte items to be aligned on 2-byte boundaries, so it will load and store `real*8`, `real*4`, and `integer*4` items 2 bytes at a time.

   6 To allow 4-byte alignment of 8-byte items, use `+A3`. The `+A3` option generates better code than `+A`. It allows 8-byte items to be aligned on 4-byte boundaries, so it will load and store `real*8` numbers 4 bytes at a time.

   Refer to the *FORTRAN/9000 Programmer's Reference* (Series 700/800) for more information.

3. Finally, you can use `allow_unaligned_data_access()` to avoid alignment errors. `allow_unaligned_data_access()` sets up a signal handler for the SIGBUS signal. When the SIGBUS signal occurs, the signal handler extracts the unaligned data from memory byte by byte.

   To implement, just add a call to `allow_unaligned_data_access()` within your main program *before* the first access to unaligned data occurs. Then link with `-lhppa`. Any alignment bus errors that occur are trapped and emulated by a routine in the `libhppa.a` library in a manner that will be transparent to you. The performance degradation will be significant, but if it only occurs in a few places in your program it shouldn't be a big concern.

   You can declare a named common block as follows:

```
        integer icnt
        common /unaligned_access_count/ icnt
```

Then you can print out the integer at the end of your program to see the extent of your trapping problem. Estimate about 0.1 msec (.0001 sec) per trap on a 720 to see if the trap handler is costing you a significant amount of time.

Whether you use alternative 2 or 3 above depends on your specific code.

The `+A` options cost significantly less per access than the handler, but they cost you on every access, whether your data is aligned or not, and they can make your code quite a bit bigger. You should use them selectively if you can isolate the routines in your program that may be exposed to misaligned pointers.

There is a performance degradation associated with 3 since each unaligned access has to trap to a library routine. You can use the `unaligned_access_count` variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use 2. If you only occasionally use a misaligned pointer, it is probably better just use the `allow_unaligned_data_access` handler. There is a stiff penalty per bus error, but it doesn't cause your program to fail and it won't cost you anything when you operate on aligned data.

The following is a an example of its use within a FORTRAN program:

```
      program ftest
      integer icnt
      common /unaligned_access_count/ icnt
      integer i1(10000), i2(10000)

      call allow_unaligned_data_access()

      call veccpy(i1(1), i2(1), 1000)
      call veccpy(i1(5000), i2(5000), 1000)
      print *,icnt
      end

      subroutine veccpy(x, y, n)
      integer n
      real*8 x(n), y(n)
      do ii=1,n
         y(ii) = x(ii)
      end do
      end
```

To compile and link this program, enter

`f77` *filename*`.f -lhppa`

This enables you to link the program with **`allow_unaligned_data_access()`** and the **`int unaligned_access_count`** that reside in **`/usr/lib/libhppa.a`**.

Note that there is a performance degradation associated with using this library since each unaligned access has to trap to a library routine. You can use the **`unaligned_access_count`** variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use the compiler option instead.

Symbolic Names

All HP-UX FORTRAN implementations allow symbolic names to be at least 255 characters long, all of which are significant.

Lowercase Letters

HP-UX FORTRAN programs can be written using lowercase letters, which is nonstandard. The FORTRAN compilers treat lowercase letters as uppercase. For example, the following symbol names are equivalent:

```
symbol_name
Symbol_Name
SYMBOL_NAME
```

Although other FORTRAN compilers typically allow lowercase letters, a few may not; you should be aware of this when porting code to other systems.

Error Conditions

Compile-time error messages are the same between the systems. All systems provide plain text error messages.

Run-time errors are similar across systems. In most cases, they are reported with the same text and number on all HP-UX systems. Some exceptions may be seen when arithmetic overflow/underflow conditions occur. On Series 300/400, the various floating-point options can cause different arithmetic error conditions than on the Series 700/800.

The order in which statements must appear in a FORTRAN program is less restrictive in HP-UX FORTRAN than in the ANSI standard. In many cases, duplicate declarations are allowed, although the result may be undefined if they are conflicting. A warning message, `array redeclaration`, will be issued.

If you will be porting to a non-HP system, then avoid using language extensions. Inserting the directive

```
$OPTION ANSI ON
```

at the beginning of your source will cause the compiler to list a warning for each use of a feature that is not a part of the ANSI 77 standard. The same effect can be accomplished by specifying `-a` on the command line.

Note Lower case letters are not supported in ANSI FORTRAN 77. If `$OPTION ANSI ON` is specified, the compiler emits a non-fatal warning once for each function in which they occur.

Character Constants

Character constants are limited to 9000 characters in length. If a longer constant is required, it can be constructed by use of the // concatenation operator. Such concatenated strings have no length restrictions.

Holleriths

Hollerith strings are limited to 2000 characters in length. To construct larger Hollerith strings at run time, you can concatenate them with the // operator.

Array Dimension Limits

While ANSI requires that FORTRAN implementations support at least 7 array dimensions, HP-UX FORTRAN permits up to 20.

Logicals

Internal representation of logical `.TRUE.` values varies across platforms as shown in Table 6-2.

Table 6-2. Representations of `.TRUE.`

Logical Type	HP-UX	Domain	VMS
`LOGICAL*1`	`0x01`	`0xFF`	`0xFF`
`LOGICAL*2`	`0x0001`	`0xFFFF`	`0xFFFF`
`LOGICAL*4`	`0x000000001`	`0xFFFFFFFF`	`0xFFFFFFFF`

In addition, the way a compiler determines whether an expression has a `.TRUE.` or `.FALSE.` value varies across platforms. Table 6-3 summarizes how various compilers determine whether an expression evaluates to `.TRUE.`. Table 6-3 also shows the type of operator used by the compiler in logical expressions. A logical operator always returns `.TRUE.` or `.FALSE.`; a bitwise operator returns a bitwise combination of the operands.

Table 6-3. Compiler Tests for `.TRUE.`

Logical Type	HP-UX	Domain	VMS
`LOGICAL*1`	`!= 0`	`< 0`	`& 0x01`
`LOGICAL*2`	`!= 0`	`< 0`	`& 0x0001`
`LOGICAL*4`	`!= 0`	`< 0`	`& 0x00000001`
Operators	Logical	Logical	Bitwise

Note

Prior to 9.0, the Series 800 FORTRAN compiler used different internal representations and tests for `.TRUE` than shown above.

The `$HP9000_800 LOGICALS` directive tells the compiler to use the same representation and test for `.TRUE.` as was used by the pre-9.0 Series 800 compiler. The `+800` compile line option turns on that alternative logical mode, as well as turn on other pre-9.0 Series 800 compatibility features.

Other vendors' logical modes can be enabled with other flags. For VAX logicals, use `$NOSTANDARD LOGICALS`, `+E2`, or `+e`. For Domain logicals use `$APOLLO LOGICALS` or `+apollo`.

Recursion

One major feature of HP's version of FORTRAN is that it supports recursion. This means that variable storage for subroutines and functions is dynamic. Thus, variables in subprograms do not retain their values between invocations, unless they are in common blocks or are saved with the `SAVE` statement or the `-K` compile line option.

Data File Compatibility

Because all HP-UX FORTRAN implementations use similar run-time I/O libraries and data types are compatible on all HP-UX systems, unformatted data files created on one system can be read on any other, *if no records or structures are used*. If records and structures are used, they can be accessed properly only if the appropriate data alignment directive is specified: `$HP9000_300 ALIGNMENT`, `$HP9000_800 ALIGNMENT`, or `$NOSTANDARD ALIGNMENT` (the last directive is on the Series 300/400 only). The ability to read unformatted data files across systems is very useful since unformatted I/O is typically the fastest data storage and retrieval mode available.

For example, the following **writer program** creates an unformatted data file `testdata`. This data file can be transported to any HP-UX system and when read, will give the same results.

```
      program testwriter
      character*1 a
      integer*2 b
      logical*2 c
      integer*4 d, ii
      logical*4 e
      real f
      double precision g
      complex h
      double complex i

      open (3,file='testdata',form='unformatted')
      do 10 ii = 1,5
         a = char(ii+33)
         b = ii
         c =  (mod(ii,2) .eq. 0)
         d = ii
         e =  (mod(ii,2) .eq. 0)
         f = ii
         g = ii
         h = cmplx(ii,ii+1)
         i = dcmplx(ii,ii+1)
         write(3) a,g,b,g,c,g,d,g,e,g,f,g,h,i
10    continue
      end
```

Here is the **reader program**:

```
       program testreader
       character*1 a
       integer*2 b
       logical*2 c
       integer*4 d, ii
       logical*4 e
       real f
       double precision g,g1,g2,g3,g4,g5
       complex h
       double complex i

       open (3,file='testdata',form='unformatted')
       do 10 ii = 1,5
        read(3) a,g1,b,g2,c,g3,d,g4,e,g5,f,g,h,i
            print *,a,b,c,d,e,f,g,g1,g2,g3,g4,g5,h,i
   10  continue
       end
```

The output of the `testreader` program will be the same on all HP-UX systems.

If you use records and structures, rather than lists of variables as in the above example, then you must use an alignment directive to force the same alignment on different systems. For example, the following program writes unformatted records using Series 700/800 alignment on the Series 300/400. All record fields are aligned on "natural" boundaries (that is, address MOD size of field = 0):

```
$hp9000_800 alignment

      program recwriter
      structure /stuff/
        byte         b1, b2, b3
        integer      i
        logical*1    o
        character*4  c4
        real*8       r
      end structure

      record /stuff/ r

      open (3,file='testdata',form='unformatted')
      do i=1,5
           ...                      ! assign the fields of record r
         write (3) r                ! write the record unformatted
      end do
      end
```

The program to read the records back in is shown below. Notice that the `$hp9000_800 alignment` directive must be included on the Series 300/400 in the reading program to ensure that the records are read with the proper data alignment.

```
$hp9000_800 alignment

      program recreader
      structure /stuff/
        byte          b1, b2, b3
        integer       i
        logical*1     o
        character*4   c4
        real*8        r
      end structure

      record /stuff/ r

      open (3,file='testdata',form='unformatted')
      do i=1,5
         read (3) r              ! read the record unformatted
         ...                     ! do whatever with the record
      end do
      end
```

6

For details on the use of the alignment directives and compile line options, refer to the FORTRAN documentation for your system.

Formatted data files created on any HP-UX system are also readable on all HP-UX systems in the same manner.

Parameter Passing

By default, HP-UX FORTRAN parameters are passed by reference. That is, the address of the actual parameter is passed to the called procedure. The procedure then dereferences the address to access the actual parameter.

For character strings, HP-UX FORTRAN passes the string length as a "hidden" parameter at the *end of the parameter list*. The string is passed by reference, and the string length parameter is hidden in the sense that the programmer does not have to specify it.

Note Code compiled on Series 800 prior to 9.0 passes character strings by descriptor.

You can change the parameter passing mode by using one of the nonstandard built-in parameter passing functions described below. These are useful for calling routines written in other languages which have different parameter passing conventions. You can use the built-in functions in a parameter list (for example, `CALL SUBR( %VAL(` *actual_arg* `))`) or in the `ALIAS` directive. If specified in both places, the parameter list in the `CALL` or function reference takes precedence over the `$ALIAS` specification.

The `$ALIAS` directive is supported on all HP-UX implementations. You can specify the parameter passing modes in the optional argument list portion of the directive (for example, `( %VAL, %REF, %VAL )`). On the Series 700/800, the directive also supports the C, Pascal, and COBOL language keywords to specifically set the parameter passing mode for that particular language. See the *HP9000 FORTRAN Programmer's Reference* for details on the `$ALIAS` directive.

The built-in parameter passing functions are:

`%VAL`	specifies that the value of the actual argument is to be passed to the called procedure. This can be used with all types of arguments on the Series 700/800. However, when used with actual arguments of a procedure, it has no effect; that is, a pointer to the procedure is still passed unless pre-9.0 Series 800 argument passing is in effect by using `+800` or `$HP9000_800 ON`. In this case, `%VAL` disables the pre-9.0 Series 800 style of passing the address of a pointer to the procedure for this argument. `%VAL` can only be used with scalar arguments of 4 bytes or less on the Series 300/400.
`%REF`	specifies that the address of the actual argument is to be passed to the called procedure. This is the compiler default, except for character arguments. For character arguments, this disables the passing of the hidden length parameter.
`%DESCR`	specifies that a descriptor is passed for this argument. HP-UX descriptors consist of the address of the object followed by the length of the object. For actual arguments of a procedure, a descriptor does not contain the length, but contains another level of indirection for the procedure name: instead of a pointer to the procedure is passed. For character strings and procedure name arguments, `%DESCR` corresponds to the pre-9.0 Series 800 convention for passing these types of arguments. `%DESCR` is not supported on the Series 300/400.

Common Region Names

ANSI FORTRAN 77 prohibits the use of the same name for a common region and a subprogram. However, some implementations do permit this overlapping as an extension. Programs that use the same name for a common region and a subprogram will not run correctly on any Series unless the `RENAME_COMMON` directive is specified. This applies to intrinsic function names also.

Note When using the `RENAME_COMMON` directive, the compiler changes the external name of the common region. Programs which interface to other languages and which depend on common regions for communication will not work unless the `$ALIAS` directive is also used to modify the external name.

Vector Instruction Set Subroutines

Listed below are Vector Instruction Set (VIS) subroutines that are supported on HP-UX. On all HP-UX implementations, they are found in `/usr/lib/libvis.a`, which you must link explicitly with the `-lvis` option to the linker or the FORTRAN compiler. These routines may not be available on other vendors FORTRANs.

DVABS	DVNRM	VADD	VNRM
DVADD	DVPIV	VDIV	VPIV
DVDIV	DVSAD	VDOT	VSAD
DVDOT	DVSDV	VMAB	VSDV
DVMAB	DVSMY	VMAX	VSMY
DVMAX	DVSSB	VMIB	VSSB
DVMIB	DVSUB	VMIN	VSUB
DVMIN	DVSUM	VMOV	VSUM
DVMOV	DVSWP	VMPY	VSWP
DVMPY	VABS		

Compile Line Options

The HP-UX FORTRAN compilers support different compile line options. Table 6-4 shows the options that differ on the systems. Options that are the same on both systems are not listed here. See your system's FORTRAN documentation and *f77*(1) for details on compile line options.

Table 6-4. Differences in FORTRAN Compiler Command Lines

Option	Effect	Difference
`+A`	Force 2-byte data alignment	All Series support `+A`, `+A3`, and `+A8`; Series 300/400 only also supports `+AN`.
`+bfpa`	Floating-point option	Series 300/400 only.
`+DA1.0`	Optimize for Series 800 architecture and instruction set. Or, use `DA8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only.
`+DA1.1`	Optimize for Series 700 and some Series 800 architectures and instruction sets. Or, use `DA7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only.
`+DS1.0`	Optimize for Series 800 instruction scheduling. Or, use `DS8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only.
`+DS1.1`	Optimize for Series 700 and some Series 800 instruction schedulings. Or, use `DS7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only.
`+ffpa`	Floating-point option	Series 300/400 only.
`+FP`*string*	Initializes the flags that specify how floating-point exceptions should be trapped.	Series 700/800 only.
`+I`	Instructs the compiler to prepare object code for profiling.	Series 700/800 only.
`+M`	Floating-point option	Series 300/400 only.
`+O`*n*	Specify optimization level	Semantics differ between systems. Refer to *f77*(1) for specific differences.

Table 6-4.
Differences in FORTRAN Compiler Command Lines (continued)

Option	Effect	Difference
`+P`	Directs the compiler to use profile information to guide code generation and profile-based optimization.	Series 700/800 only.
`+P1`, `+P2`	Invoke specific optimizer phase	Series 300/400 only.
`+T`	Procedure traceback	Series 700/800 only.

Note The `+O`*n* option replaces `-O1` and `-O2` on the Series 800.

Compiler Directives

6 Most compiler directives are portable across the HP-UX FORTRAN implementations; if not, the directive is ignored and a warning message is issued.

Since compiler directives themselves are non-standard, it is recommended that if you need to use them, you place them in a directives file and use the `+Q` compile-line option. The `+Q` compile-line option causes the compiler to look for compiler directives in the file specified before compiling the FORTRAN program. In this way, the compiler directives are not included in the source file.

Directives Only on Series 300/400

The `INLINE` directive is the only one exclusively on Series 300/400.

Directives Only on Series 700/800

CHECK_OVERFLOW	LIST_CODE	ASSEMBLY	CHECK_FORMAL_PARM
CODE_OFFSETS	LOCALITY	SEGMENT	CHECK_ACTUAL_PARM
INIT			

OPTIMIZE

When you use OPTIMIZE on Series 300/400, the command line must also specify -O, +O1, +O2 or +O3.

SAVE_LOCALS

On Series 300/400, the SAVE_LOCALS directive causes saved variables to be initialized to zero; it does *not* initialize variables on Series 700/800.

Temporary Files ($TMPDIR)

Series 300/400 compilers produce a number of intermediate temporary files usually on /tmp or /usr/tmp, for their private use during the compilation process. These files are normally invisible to you because they are created and removed automatically. If, however, your system is tightly constrained for file space, the requirements for these files may exceed the space available. By assigning another directory name to the TMPDIR environment variable you can redirect these temporary files. See *f77*(1) for details.

The Series 700/800 compiler does not create temporary files, so TMPDIR is ignored, except for .F files where a temporary file is created for the output from cpp.

lintfor: Extended Syntax Checker

All HP-UX implementations provide the lintfor syntax checker which can be used to find some nonportable constructs. This command does a static analysis of a source program to find nonstandard or dubious programming. lintfor does not produce object code; it only does a syntax check. For details on the use of lintfor, see *FORTRAN/9000 Programmer's Guide.*

ratfor Support

HP-UX supports ratfor, a "rational" FORTRAN dialect preprocessor. ratfor translates a superset of FORTRAN, adding certain control constructs patterned after statements found in the C language to the standard FORTRAN source code. Since ratfor source code is widespread throughout the industry, HP-UX provides this preprocessor on all implementations. However, it is unlikely that rewriting existing FORTRAN into ratfor will be to your advantage.

ASA Carriage Control

Another FORTRAN utility that is useful is **asa**, a filter that interprets ASA carriage control characters. These carriage control characters will merely be printed as opposed to being used for carriage control on HP-UX unless **asa** is used during the execution of the FORTRAN program.

For example, consider the following FORTRAN program:

```
      program testasa
C     Unit 6 is preconnected to stdout on HP-UX.
C     Note that some terminals may disregard printer
C     control characters.
      write(6,100)
      write(6,200)
      write(6,300)
      write(6,400)
      write(6,500)
100   format(" A blank line should precede this line.")
200   format("0This line should be double spaced.")
300   format("1This line should come out on a new page.")
400   format(" This is a ")
500   format("+          concatenated line.")
      end
```

After compiling, if this program is executed without **asa** using the command

```
a.out | lp
```

the output to the printer will be

```
 A blank line should precede this line.
0This line should be double spaced.
1This line should come out on a new page.
This is a
+          concatenated line.
```

On the other hand, if **asa** is included in the pipe as a filter as in the following command:

```
a.out | asa | lp
```

the output to the printer will be

```
A blank line should precede this line

This line should be double spaced.
{new page here}
This line should come out on a new page.
This is a concatenated line.
```

Checking for Standards Compliance

If you have the HP Advise product, you can also check for FORTRAN standard compliance using the **apex** command.

Porting FORTRAN Programs between Series 300/400 and Series 700/800

The FORTRAN compiler has an option, +800, which allows you to compile in a compatible mode with the pre-9.0 Series 800 FORTRAN compiler. Due to the performance penalty incurred, using +800 is only recommended for

- Compiling prior Series 800 FORTRAN code that depends on features of the earlier implementation such as its logical representation or argument passing conventions, or,
- Linking with objects compiled with the pre-9.0 Series 800 FORTRAN compiler; for example, you may need to link with a third party library that has not been recompiled for 9.0.

If you do not specify +800, by default your code will be compatible with all previous pre-9.0 FORTRAN code from the Series 700 and Series 300/400. (The remainder of this chapter assumes you are using this default mode.) When porting from a Series 300/400 to Series 700/800, you should use this default mode. If you move programs to the Series 700 from the pre-9.0 Series 800, no re-compilation is required. However, to take advantage of new hardware instructions on the Series 700, you must recompile. If you do recompile, you should use the +800 option only if one or both of the situations listed in the above paragraph exist.

For the most part, Series 700/800 FORTRAN is identical to Series 300/400 with these notable exceptions:

- The Series 700/800 has different alignment rules than the Series 300/400.
- There are architectural differences that may be exposed if you use inherently unportable practices.
- There are differences in arithmetic due to the different processors.
- There are some language feature differences.
- There are development environment differences. These do not affect program portability, per se, but may require changes in makefiles or development scripts.

The following sections elaborate on these differences.

Data Alignment

The Series 700/800 processor generally requires that data be aligned on boundaries equivalent to their size, whereas the Motorola architecture of the Series 300/400 has less restrictive alignment. (For details on data types and alignments, see Table 6-1.) There are two scenarios in which this difference may cause portability problems in FORTRAN programs:

- Using `EQUIVALENCE` or `COMMON` statements in a way that allows references to a variable that may be incompatible with the type of that variable, or making assumptions about the way data is laid out in a common block.
- Using `COMMON` to access a C structure from a FORTRAN routine (or vice versa).

The `EQUIVALENCE` statement can cause portability problems due to different architectures' varying requirements for data alignment. In short, `EQUIVALENCE` makes it possible to request that data be arranged in a manner incompatible with the processor architecture. For example, the following code example compiles and runs correctly on the Series 300/400, but fails to compile on the Series 700/800.

```
program equiv1
real*4 a(100)
real*8 x,y
equivalence (a(1), x) (a(2), y)
end
```

Attempting to compile this on the Series 700/800 produces the following error message because of the differing alignment requirements for `real*8` variables; the program tries to align an 8-byte datum on a 4-byte boundary:

```
Declaration error on/above line 6 of equiv1.f; for x;
bad alignment forced by equivalence
```

The `COMMON` statement can cause similar problems when used to simulate C structures in FORTRAN. If FORTRAN and C arrange the data differently, then you will have problems when passing the common block to a C function which expects a pointer to a structure. This should not be a problem when porting from the Series 300/400 to the Series 700/800, although if you are using that common block/structure to create binary data files, those files may not be compatible between the two systems. Refer to Chapter 5 on porting C programs for details.

When investigating alignment problems, it is occasionally useful to find out exactly how the compiler is aligning a datum. The following C program can be linked with your FORTRAN program and then called with the datum of interest.

```
printaddr(name, datum)
char   *name;
char   *datum;
{
    printf("%s: 0x%8.8x, (%d)\n", name, datum, (int)datum % 4);
}
```

You can then call this function from FORTRAN like this:

```
program align
integer*4 i4(10), j4(10)
integer*2 i2(50)
common i2
equivalence (i2(1), i4(1))
equivalence (i2(22), j4(1))
call printaddr('i4(1)'//char(0), i4(1))
call printaddr('j4(1)'//char(0), j4(1))
end
```

The Series 700/800 FORTRAN compiler has a number of directives that alter the way the compiler aligns data, and consequently, the instructions used to access that data. A complete discussion of these directives is beyond the scope of this guide, but be warned that the directives that invoke non-native alignments may cause performance degradation due to sub-optimal data accesses. (For instance, having to access a 4-byte value as two 2-byte values because it's not aligned to a 4-byte boundary.)

Implementation Differences

There are a few differences between FORTRAN on the Series 300/400 and on the Series 700/800 that will only be a problem if your program uses inherently unportable techniques:

- A variable declared as a formal argument in a program unit and used following an `ENTRY` statement, but not listed as a formal argument in the `ENTRY`, behaves differently on the two systems.
- The value of an uninitialized local variable may be different on the Series 700/800 than it is on the Series 300/400.
- As discussed in Chapter 3, the execution stack on the Series 700/800 grows towards higher addresses while on the Series 300/400, it grows towards lower addresses. If you directly manipulate the stack, the same code will not work for both architectures.
- FORTRAN intrinsic routines are in different libraries on Series 700/800 than on Series 300/400. Also, the symbol in the library corresponding to a particular FORTRAN intrinsic may not have the same name. If your program makes assumptions about library internals, it will probably have to be altered to run on the Series 700/800.

Arithmetic Differences

Because the Series 700/800 have different math hardware than the Series 300/400, there are some inevitable differences in the results of some arithmetic operations:

- The Series 700/800 and the Series 300/400 implement full IEEE arithmetic. However, the Series 300/400 uses 80-bit internal representations of floating-point numbers while the Series 700/800 is limited to 64. Consequently, there may be some loss of precision on the Series 700/800.
- A few equations yield different results on the two machines:

Equation	Result on Series 300/400	Result on Series 700/800
`0**0`	0	1
`0**-1`	0	+INF
`SQRT(x), x < 0`	0	Not a number

- Floating-point overflow and some underflows on the Series 300/400 produce the error message `Floating exception - core dumped`, but on the Series 700/800, overflow produces a result of `INF`, while underflow produces `0.0`. The Series 700/800 can trap for overflow or other conditions if the `+T` option is used. See the *FORTRAN/9000 Programmer's Reference* for more information.
- It is much more difficult for a user program to do math exception handling on the Series 700/800 than on the Series 300/400. A discussion of the differences is beyond the scope of this document. (See the `ON` statement in the *FORTRAN/9000 Programmer's Reference* for more information.)

Language Feature Differences

6

Some FORTRAN language features available on the Series 300/400 are either unavailable or work differently on the Series 700/800.

- The `$NOSTANDARD ALIGNMENT` directive is not supported on the Series 700/800.
- The `ISHFTC` intrinsic will not handle arguments that are out of range.
- The constant `MAXINT` may be used as an upper loop bound on the Series 300/400, but not on the Series 700/800.

Development Environment Differences

There are some differences in the commands used to compile and link a FORTRAN program on the two systems. There are also some primarily cosmetic differences (for example, compiler error and warning messages may be worded differently). Such differences may require changes in your makefiles or development scripts. See the *FORTRAN/9000 Programmer's Reference* for details on the different options.

Porting between HP-UX FORTRAN and VMS FORTRAN

Because VMS FORTRAN has been a popular programming environment for many years, many programs have been written in VMS FORTRAN. Although most of these programs use extensions specific to this environment, Hewlett-Packard Company realizes that these programs represent a substantial software investment. Consequently, an effort has been made to understand the differences between VAX VMS FORTRAN and HP-UX FORTRAN and to provide mechanisms to make porting of these programs easier.

As is the case with most FORTRAN implementations, the most difficult areas of compatibility are in the areas of operating system interfaces, file manipulation, and input/output. To some extent, there are differences in extended language feature sets and compiler options as well.

Both the VMS and the HP-UX compilers support the full ANSI FORTRAN 77 standard and MIL-STD-1753 extensions. However, the VAX VMS compiler has evolved from ANSI FORTRAN 66, an earlier standard. It therefore supports many language features that predate the current standard. It also supports a rich set of extensions peculiar to the VMS environment. This section primarily describes the differences between the extensions to the FORTRAN 77 standard.

FORTRAN Applications without VMS System or Run-Time Library Calls

If there are no VMS system or run-time library calls, and the application is written completely in FORTRAN, using only FORTRAN I/O facilities, then the language comparison below can be consulted for the differences between the FORTRANs on these systems. In general, differences between the HP-UX and VMS operating systems will not arise in this case.

When using only FORTRAN-defined I/O, one important issue remains. If you have a VMS FORTRAN application that writes unformatted (binary) data in a file that will be read by a different FORTRAN program, then you should port both the writer and reader programs to HP-UX. If the writer program runs on VMS, and the data file is moved to HP-UX over a local area network or by magnetic tape, the HP-UX reader will not be able to correctly read the file. Both the format of the file (for example, the file header and the record headers and trailers) and the byte representations of the data, will be different between VMS and HP-UX, even though FORTRAN I/O facilities were used exclusively. The simplest way to move data is to convert it to ASCII to solve the bit representation problem, and then move it using a common format. Large arrays of bytes (like graphics pixel maps) can probably be moved without conversion to ASCII, if a common file format can be agreed upon. However, some translation will be required to make the resulting file readable as an unformatted FORTRAN file on HP-UX. In this case, consider writing the necessary conversion program to move the data to FORTRAN the first time.

The difference in hardware that exists between the VAX architecture and most other computer architectures may cause problems since FORTRAN's `EQUIVALENCE` statement and bit operations allow system-dependent coding. An application that depends on the bit representations of numbers instead of their values can compile with no errors and still produce unexpected results when run.

For example, the following program produces different results when run on VMS and HP-UX.

```
C       A program that compiles and produces different results
C       on a VAX system than on an HP system
C
        program machdep
        integer*2 i(4)
        integer*4 j(2),sum
        equivalence (i,j)
        do 10,ii=1,4
           i(ii) = ii
 10     continue
        sum = 0
        do 20,ii=1,2
           sum = sum + j(ii)
 20     continue
        print *,sum
        end
```

This example depends on the byte ordering of integers. It prints 262150 on an HP-UX system and 393220 on a VAX system.

FORTRAN Applications with VMS System or Run-Time Library Calls

To check for VMS system calls or VMS run-time library calls, search the source code for `$` using the HP-UX command `grep` or the VMS DCL command `search`. VMS system call names start with `SYS$` (like `SYS$QIOW` and `SYS$ASSIGN`) and run-time library routine names start with various prefixes including `LIB$`, `STR$`, and `SMG$`. HP-UX FORTRAN compilers accept procedure names with the `$` character so problems do not become evident until the linker fails to resolve the references to these VMS routines.

Note VMS system calls can also be abbreviated as starting simply with `$`, instead of `SYS$`. So, for example, `SYS$ASSIGN`, can be written as `$ASSIGN`.

Once you have located VMS system or run-time library calls, you can do any of the following:

- Write emulation or **onionskin** routines in C or FORTRAN that use HP-UX system and library calls. If you use emulation or onionskin routines written in C (the easiest way to get to the HP-UX routines), you'll probably need to change the VMS names since HP-UX C compilers will not accept the `$` character in names. (Or alternatively, use `$ALIAS`.) A programmer undertaking this task will need to be quite familiar with Sections 2 and 3 of the *HP-UX Reference* in addition to being knowledgeable about VMS and the application program.
- Modify the source to use HP-UX routines directly. (You should note that there is very little chance that this is as simple as finding the HP-UX routine that exactly matches the functionality of the VMS one.)
- Use a third party product to assist you with the VMS to HP-UX port.

An example is a program that needs to read input from a graphical input device without waiting for a standard terminating character. FORTRAN's `READ` statement will not suffice here. The VMS solution uses `SYS$ASSIGN` to allocate a channel number and then uses `SYS$QIOW` to perform low level reads and writes to the device. Similar functionality on HP-UX can be obtained by using `open(2)` to return a file descriptor instead of `SYS$ASSIGN`'s channel number and by using `ioctl(2)`, `read(2)`, and `write(2)` to set up character I/O and perform the I/O operations.

Graphics and Windows

FORTRAN does not define any graphics functionality. The most common graphics applications will either include a "graphics driver" written in FORTRAN that sends Tektronix^{TM} (or some other vendor) escape sequences over RS-232, or it will reference an object code library for a proprietary graphics display system. In the case of the RS-232 type driver, the code will usually port directly and can be used to drive the same type of display connected to your HP-UX system with RS-232.

If the application uses a proprietary display or you wish to use HP's family of graphical devices, you will need to convert the graphics calls into HP's Starbase library calls. For all but the simplest graphics needs, this will probably involve some redesign of the graphics part of the application. In some cases, a nearly one-to-one translation of graphics calls may suffice.

HP-UX and VMS (along with several other vendors' systems) now share a common windowing system based on the X11 Window System from MIT. This brings higher compatibility for windowing and simple graphics functionality to these systems. However, since the X library interface uses a C language definition, it requires data types and calling conventions not normally found in FORTRAN 77 but available as extensions in VMS FORTRAN. Consequently, an X application written in VMS FORTRAN will not compile and run on all HP-UX implementations. HP provides some FORTRAN bindings for X that will make this task easier, but source modifications will be necessary.

Comparisons of Core Language Features

The next several subsections list VAX VMS implementation features. Each item is also supported on HP-UX, unless otherwise stated. Also, unless otherwise stated, each HP-UX feature is supported on all implementations.

Character Sets

- Lower case ASCII letters are converted into their upper case counterparts except within Hollerith or quoted strings. This is the default for VMS and HP-UX. HP-UX FORTRAN also provides the +U compile line option for making lower case and upper case ASCII characters distinct.

- TAB characters have the same behavior on VMS and HP-UX implementations between columns 1-72.
- Quotation marks ("), underscore (_), exclamation point (!), dollar sign ($), ampersand (&), and percent sign (%) are supported.
- CONTROL-L within source code produces a new page in the source listing.
- Left and right angle brackets (< and >) are used only to delimit variable expressions within formats. This is also supported on HP-UX.
- The VMS FORTRAN Radix-50 character set is not supported on HP-UX.

Control Statements

- The `DO` ... `WHILE` control construct is supported.
- The `DO` ... `END DO` control construct is supported.
- Forcing FORTRAN 66 semantics on `DO` loop evaluation by requiring a minimum of 1 iteration of the loop can be enabled via a compiler option.
- Jumps into `IF` blocks or `ELSE` blocks are allowed.
- Extended-range `DO` loops are permitted. That is, jumps out of a `DO` loop to other executable code are allowed as long as control eventually returns back to within the `DO` loop by means of an unconditional `GOTO` .

Data Types and Constant Syntaxes

- The `BYTE` data type is supported.
- The `LOGICAL*1`, `LOGICAL*2`, `LOGICAL*4` data types are supported.
- The `INTEGER*2` and `INTEGER*4` data types are supported.
- The `REAL*4`, `REAL*8`, `COMPLEX*8` and `COMPLEX*16` data types are supported.
- `REAL*16` is supported.
- The `DOUBLE COMPLEX` data type (synonym for `COMPLEX*16`) is supported.
- Octal constants of the form `O'ddd'` or `'ddd'O` are supported.
- Hexadecimal constants of the form `Z'ddd'` or `'ddd'X` are supported.
- Octal and hexadecimal constants are considered to be "typeless" and may be used anywhere a decimal constant may be used.

- Hollerith is supported. It is also supported on all HP-UX implementations but in different ways. On Series 300/400 HP-UX compilers, Hollerith is treated internally as a synonym for a quoted character constant. On Series 700/800, Hollerith constants are considered "typeless" and may be used anywhere a decimal constant may be used.

Note For pre-9.0 FORTRAN on Series 700/800, Hollerith constants were treated as integers.

- Character constants have a maximum length of 2000 characters on VMS. Refer to "Character Constants" earlier in this chapter for HP-UX limits.
- `RECORD` and `STRUCT` data types are supported. They are also supported on HP-UX. Default alignment differs from VMS. On the Series 300/400 only, use the `$NOSTANDARD ALIGNMENT` directive to force VMS alignment.
- VMS FORTRAN octal constants of the form `"`*ddd* are not supported on HP-UX.
- VMS FORTRAN `REAL*8` (`D_floating`) and `COMPLEX*16` (`D_floating`) are not supported on HP-UX.

General Statement Syntax and Source Program Format

- An exclamation point can be used for end-of-line comments.
- `D` is recognized in column 1 for debug lines.
- `INCLUDE` *filename* is allowed for including source statements.

Note This statement should not be confused with the `$INCLUDE` compiler directive.

- On VMS, sequence numbering is allowed in columns 73-80. VMS ignores sequence numbers. HP-UX ignores anything in columns > 72.
- Continuation lines are different on HP-UX than VMS. On HP-UX, 99 continuation lines are allowed.
- An initial tab followed by a non-zero digit is interpreted as a continuation line.

- Up to 132 columns can be made significant under a VMS compiler option; this is not supported on HP-UX unless the `+es` option is used.

Note Do not compile code which has sequence numbers with `+es`; errors will result.

- `DATA` statement location is handled differently on HP-UX than VMS. `DATA` statements can be interspersed with executable statements on all Series provided the `-K` or `+e` compile line option is specified.
- Alternate forms of data type length specification are allowed, for example:

  ```
  INTEGER FOO*4
  ```

- Variables may be initialized when they are declared, for example:

  ```
  INTEGER IARRAY(3) /4,5,6/
  ```

- `DATA` statements may be used for initialization of common block variables outside of `BLOCK DATA` subprograms.
- Octal, hexadecimal and Hollerith constants are allowed within `DATA` statements.
- Octal, decimal, hexadecimal, and Hollerith constants may be used within `DATA` statements to initialize `CHARACTER*1` variables.
- Two arithmetic operators may be consecutive if the second is a unary operator. Be aware that precedence may be changed; for example:

  ```
  I = IA + -3
  ```

- `INCLUDE` *library_module* for including selected library routines is supported on VMS but not on HP-UX.

Input/Output Statements

- `DECODE` and `ENCODE` are supported.
- Namelist-directed I/O is supported.
- List-directed internal I/O is supported.

- Files opened for `DIRECT` access can have sequential I/O operations performed.
- The `TYPE` statement is supported.
- An optional comma (`,`) is allowed to precede the *iolist* within a `WRITE` statement; for example:

  ```
  WRITE(6, 100) , A, B
  ```

  is equivalent to

  ```
  WRITE(6, 100) A, B
  ```

- The `RECL=` I/O specifier for an `OPEN` statement will be converted to `INTEGER` if it is not already.
- The `UNIT=` and `REC=` I/O specifiers will be converted to `INTEGER` if they are not already.
- Full variable-format expression support is provided on VMS; on HP-UX, it is supported on all implementations, if the **+e** or **+E6** compiler option is specified.
- The `RECL=` I/O specifier counts words on VMS, but bytes on HP-UX.
- Unlike VMS, the `FILE=` I/O specifier must be `CHARACTER` on HP-UX.
- The `ACCESS='APPEND'` specifier for the `OPEN` statement is supported.
- The `MAXREC=`*n* specifier for the `OPEN` statement is supported.
- Auto-opening of files is supported.
- `O` and `Z` field descriptors are supported.
- The `$` edit descriptor is supported.
- VMS permits the `H` field descriptor to be used with `READ` as well as `WRITE`; it can be used only with `WRITE` on HP-UX.
- Unlike on VMS, on Series 700/800, the *n*`H` descriptor is treated as *n*`X` when used with `READ`.
- The `Q` edit descriptor is supported.
- Default field descriptors are supported.

- $ and ASCII NUL carriage control characters are supported; they are not supported on HP-UX.
- The ACCEPT statement is supported.
- DEFINE and FIND are supported on VMS but not on HP-UX.
- DELETE, REWRITE, and UNLOCK statements are supported on VMS; on HP-UX, they are only supported on Series 700/800.
- Key-field and key-of-reference specifiers are supported on VMS and on HP-UX on Series 700/800 only.
- The VMS concept of indexed file access is supported on Series 700/800 only, provided you also install an appropriate 3rd party ISAM product.
- The HP-UX FORTRAN compiler emits warning messages for unsupported VMS keywords and I/O specifiers.
- The following VMS keywords are supported only on Series 700/800, while Series 300/400 gives a nonfatal warning and ignores the keyword clause:

  ```
  KEY
  KEYED
  KEYID
  RECORDTYPE
  SHARED
  ```

- HP-UX implementations give a nonfatal warning and ignore the keyword clause for the following VMS keywords:

  ```
  ASSOCIATEVARIABLE   DISP          NOSPANBLOCKS
  BLOCKSIZE           DISPOSE       ORGANIZATION
  BUFFERCOUNT         EXTENDSIZE    RECORDSIZE
  CARRIAGECONTROL     INITIALSIZE   USEROPEN
  DEFAULTFILE
  ```

- A comma (,) can be used to separate numeric input data to avoid having to blank fill (i.e. short field termination).
- Extraneous parentheses are permitted around I/O lists for READ and WRITE statements on VMS and all HP-UX Series; for example:

  ```
  WRITE (6, 100) (A, B, C)
  ```

Intrinsic Functions

- VMS FORTRAN SIZEOF is supported on HP-UX with the +e or +E1 compile line option or the $NOSTANDARD SYSTEM directive.
- MIL-STD-1753 intrinsics ISHFT, ISHFTC, IBITS, BTEST, IBSET, and IBCLR are supported.
- The MIL-STD-1753 subroutine MVBITS is supported.
- Transcendental intrinsics that take arguments in degrees are:

ACOSD	ATAND	DASIND	DCOSD	SIND
ASIND	COSD	DATAN2D	DSIND	TAND
ATAN2D	DACOSD	DATAND	DTAND	

- The following VMS-specific intrinsics are supported on HP-UX:

AIMAX0	CDSQRT	IIBSET	IISHFTC	INOT	JIOR	JMIN1
AIMIN0	DFLOAT	IIDIM	IISIGN	JIABS	JISHFT	JMOD
AJMAX0	DFLOTI	IIDNNT	IIXOR	JIAND	JISHFTC	JNINT
BITEST	DFLOTJ	IIEOR	IMAX0	JIBITS	JISIGN	JNOT
BJTEST	DREAL	IIFIX	IMAX1	JIDIM	JIXOR	ZEXT
CDABS	IIABS	IINT	IMIN0	JIDNNT	JMAX0	
CDEXP	IIAND	IIOR	IMIN1	JIEOR	JMAX1	
CDLOG	IIBITS	IISHFT	ININT	JINT	JMIN0	

- The following VMS-specific intrinsics to support the REAL*16 data type are supported on HP-UX:

IIQNINT	QASIND	QCOSH	QLOG	QSIGN
IJQNINT	QASINH	QDIM	QLOG10	QSIN
IQINT	QATAN	QEXP	QMAX1	QSIND
IQNINT	QATAN2	QEXT	QMIN1	QSINH
QABS	QATAN2D	QEXTD	QMOD	QSQRT
QACOS	QATAND	QFLOAT	QNINT	QTAN
QACOSD	QATANH	QFLOTI	QNUM	QTAND
QACOSH	QCOS	QFLOTJ	QPROD	QTANH
QASIN	QCOSD	QINT		SNGLQ

- VMS system subroutines (DATE, EXIT, IDATE, TIME, SECNDS, RAN) are supported with the +e or +E1 compile line option or the $NOSTANDARD SYSTEM directive. These routines are not compatible with HP-UX system functions of the same name. VMS FORTRAN ERRSNS is not supported on HP-UX.

Specification Statements

- IMPLICIT NONE turns off default type rules for variables.
- The following limited number of intrinsic functions are allowed to be used within the PARAMETER statement to define constants:

ABS	CONJG	IAND	IMAG	LGE	LLT	MOD
CHAR	DIM	ICHAR	IOR	LGT	MAX	NINT
CMPLX	DPROD	IEOR	ISHFT	LLE	MIN	NOT

  In this context, arguments to these functions must be constants.
- There is support for the alternate form of the PARAMETER statement with different semantic connotations. For example:

  ```
  PARAMETER ISTART = 3
  ```
- Symbolic constants may be used in run-time formats.
- The VIRTUAL statement is supported.
- Multidimensional arrays may be specified with only one subscript within EQUIVALENCE statements.
- The VOLATILE statement is supported.
- On VMS but not on HP-UX, symbolic constants may themselves be used to define COMPLEX symbolic constants within a PARAMETER statement.
- The NOF77 interpretation of the EXTERNAL statement (non-ANSI semantics) is supported on VMS but no on HP-UX.

Subprograms

- ENTRY points in a CHARACTER function must all be of type CHARACTER, but can have different length specifications, as for example:

```
CHARACTER*10 FUNCTION CHARFUNC(A)
    ...
CHARACTER*5 CHFUNC
    ...
ENTRY CHFUNC() ! Not ANSI but allowed on HP-UX and VMS.
    ...
END
```

- Octal or hexadecimal or Hollerith constants are treated as typeless constants when passed as actual arguments; the corresponding formal parameter can be of any type as long as the length of the formal is explicit (that is, not CHARACTER*(*)) and the length of the constant actually matches the length of the formal type.
- %loc is recognized as a built-in function to compute the internal address of a datum.
- %val, %ref, and %descr are supported within actual argument lists. The $alias compiler directive provides the same functionality and it is supported on all HP-UX implementations.
- Implementations recognize the alternate syntax for specifying the type of functions within a function declaration; for example:

```
INTEGER FUNCTION FOO*2(X, Y)
```

- Calls to subprograms can have "missing" actual arguments whose positions are indicated by a comma (,). The compiler implicitly assumes that the actual argument value is 0 and it is passed by value. For example:

```
X = FOO(,Y)
```

  is equivalent to

```
X = FOO(%VAL(0), Y)
```

- &label can be used in place of *label when specifying alternate returns.
- The controlling expression for an alternate return will be converted to INTEGER if necessary.

Symbolic Names

- Regarding symbolic names maximum length, VMS allows 31 characters within symbolic names, all significant. All HP-UX implementations allow at least 255 characters within symbolic names, all significant.
- Underscores (_) and dollar signs ($) in names are allowed.

Type Coercions

- Arithmetic operations involving both `COMPLEX*8` and `REAL*8` elements are computed using `COMPLEX*16` arithmetic on all VMS and HP-UX implementations.
- The numeric operand of a computed `GOTO` statement will be converted to an `INTEGER`, if it is not already, on all VMS and HP-UX implementations.
- Character substring specifiers may be non-integer. They are implicitly converted to integer by truncation.
- Unlike on VMS, non-integer array bound and subscript expressions will be converted to `INTEGER` by truncation on all HP-UX implementations.
- Character constants can be used in a numeric context; they are interpreted as Hollerith. Character constants and Hollerith are synonymous.
- On VMS, logical operands can be used like numeric operands, and vice versa. Logical operands can appear in arithmetic expressions and numeric operands can appear in logical expressions on all HP-UX implementations.

Miscellaneous

- `.XOR.` and `.NEQV.` are functionally equivalent operators.
- Null strings are allowed in character assignments. For example:

```
c = '';
```

- VMS represents `.FALSE.` by 0 and `.TRUE.` by −1. Logical representation on HP-UX is 0=FALSE and 1=TRUE by default. All Series can specify the **+E2** command line option or the `$NOSTANDARD LOGICALS` directive to cause the compiler to generate the VMS representations for logicals.

Data Representations in Memory

The internal allocation of memory for variable storage is primarily of interest in situations where:

- Large amounts of local data are required.
- When equivalencing the same storage locations with different data types.

Large Amounts of Local Data

You should have few problems porting programs that require large local data storage on to HP-UX. On HP-UX, data storage is limited only by the system limits for the maximum run-time stack size and maximum process size. These limits are pre-set to large default values and are further configurable by your system administrator. See the *How HP-UX Works: Concepts for the System Administrator* (all Series) for further details.

Equivalencing of Data

A problem with memory allocation usually involves equivalencing of data. In general, VMS data types take the same number of 8-bit bytes as their HP-UX counterparts. The internal representations of logical, integer, floating-point, Hollerith and character data types are not necessarily the same, however, and programs that depend on them must be modified.

Another problem with equivalenced data is that the alignment restrictions on the various data types differs between VMS and HP-UX and between the various HP-UX architectures. VAX VMS will permit a datum to begin on an arbitrary byte boundary, whereas HP-UX systems generally require that multibyte data types be aligned in memory on specific boundaries. See Table 6-1 for specific alignment requirements for the different architectures on HP-UX. The FORTRAN compilers on HP-UX normally allocate data storage to conform to alignment restrictions automatically. When using the `EQUIVALENCE` statement to force the overlay of different data types, however, the compilers do not have the freedom to allocate memory according to their own alignment rules. If an `EQUIVALENCE` class forces an illegal alignment, HP-UX compilers will report an error at compile time and refuse to generate further code.

Multibyte data types require a minimum of 2-byte alignment on HP-UX. For performance reasons, 4- or 8-byte data types are normally further restricted to 4- or 8-byte alignment. If it is necessary to use the minimum 2-byte alignment because of **EQUIVALENCE** statement structure, both Series 300/400 and Series 700/800 have a **+A** compile line option. In addition, all Series have an **HP1000 ALIGNMENT ON** inline compiler option that will cause data storage to use the minimum 2-byte alignment for multibyte data types. There are performance penalties when these options are in effect: memory references to minimally aligned data can slow 0-20% on a Series 300/400 and 0-200% on a Series 700/800 when these options are used. Program units that share common areas, or that make subroutine or function calls between them, should be compiled consistently with respect to the alignment options.

For example, the following program will not compile on either the Series 300/400 or the Series 700/800 without special alignment instructions:

```
program bench
integer*2 i2, j2
real*8 a(1024), b(1024), c(1024)
common i2, a, b
integer*2 jarray(10)
equivalence (jarray(1), i2), (jarray(2), a(1))
end
```

If **+A** or **$HP1000 ALIGNMENT ON** is specified on all Series, the above program will compile and produce the same results.

VMS-style records are supported on Series 300/400 and 700/800 FORTRAN. This makes interlanguage communication easier. If VMS alignment of structure members is required on Series 300/400, specify the **$NOSTANDARD ALIGNMENT** option. This option has no effect on the byte ordering of data within structure fields. See Table 6-1 for specific alignment requirements for Series 300/400 and 700/800.

The Effects of Recursion on Local Variable Storage

Recursion, or the ability of a subprogram to call itself directly or indirectly, is a powerful programming tool that has been implemented in all HP-UX FORTRAN compilers. It is an extension to ANSI FORTRAN and it is not available on VMS. Under normal circumstances, a non-recursive VMS program should see no effect from the recursive capabilities of an HP-UX compiler. There are, however, some attributes of the implementations of recursion that may give you a surprise if your program depends on non-ANSI features.

Inherent in a compiler supporting recursion is the introduction of a run-time stack which contains activation records for each invocation of a subprogram. During the execution of your program, an activation record is constructed on the run-time stack when a subprogram is entered and it is destroyed when the subprogram is exited. During the activation of this subprogram, all local data is normally stored within this record. The compiler allocates a location for each local variable within the activation record relative to the beginning of the record. All operations that relate to that variable will use this relative address even though the actual address of the beginning of the activation record is not known until run time and in fact, depending on the order of subprograms being executed, the location of various activation records for the same function may vary in absolute location on the stack as the program executes. Since the locations of the activation records themselves may vary, so may the locations of the local data storage within them.

Many non-recursive implementations of FORTRAN do not use a relative addressing scheme; rather, they simply assign a permanent absolute address for a datum that is to be used throughout the execution of the program. The effect is as though the variable had been designated as a `SAVE` variable; once a value has been assigned to the variable, it remains with that variable until another assignment. Neither ANSI nor HP-UX supports this behavior except for variables that are explicitly saved or in common. If a subprogram has an uninitialized variable on an HP-UX FORTRAN implementation, the initial value is random. It will in general not be the value left when the subprogram was last executed and exited. The effect to the program may be unpredictable.

Some older programs have been written with the assumption that an uninitialized local variable is implicitly initialized to 0 as execution begins. Such initialization is not supported by ANSI or HP-UX. Your program should not rely on this behavior since it will invariably become a subtle defect sometime during the life of the program. You can overcome this problem by using `$INIT ON` on Series 700/800. ANSI also does not support the implicit initialization of a common region; however, HP-UX, as a feature of its implementation, does initialize common regions to 0 unless otherwise initialized via `DATA` statements.

In most cases, programs that rely on the above assumptions do so unintentionally, since they seem to work correctly on the system where they were developed. It is only when they are ported and the assumptions fail that it is apparent that *something* is wrong. The usual indications of a problem involving these assumptions is that the problem program appears to be nondeterministic. That is, it seems to give different results (or errors) at different times for the same data or it suddenly crashes on data that works on the original architecture.

Finding bugs of this type is a tough problem as are most errors of omission. On HP-UX, there are some tools that may be useful. First, the `-K` compiler option causes static memory allocation for local variables. This has the effect of making all local variables `SAVE` variables and it forces an implicit initialization of these variables to 0. If the program behaves differently with `-K` than without it, chances are good that somewhere there is at least one variable that's improperly initialized. Specifying `-K` during compilation typically has a small effect on program performance (0 to 5% degradation). Since data is statically stored using this option, your program will have a larger disk image as well. Also, the global optimizer (enabled when `-O` is specified on the command line) will print a warning on `stderr` for most uninitialized variables. Finally, you can use the cross referencing option to help look for uninitialized variables.

Resolving System Name Conflicts

Occasionally, when porting a program from the non-HP-UX environment, a user-defined subroutine or function name will conflict with a system routine name or a library function name. The result may be inexplicable behavior or a program crash. If you suspect a problem in this area, you can specify the `-U` compiler option on all HP-UX FORTRAN implementations. This option forces the compiler to generate external names in upper case, regardless of how they are declared. Since most system routine names and library names contain at least one lower case letter, name conflicts can often be avoided. In cases where this does not work, you may have to rename the routine that is causing the system name conflict. You can also use the `+ppu` compiler option.

File Names

VMS expects file names with suffixes (e.g., `FOR015.DAT`), while HP-UX does not (e.g., `for015`).

VMS file names are always upper case; HP-UX file names, on the contrary are usually lower case.

Predefined and Preconnected Files

VMS predefines several logical file names that the operating system has associated with particular file specifications. HP-UX, since it supports FORTRAN as one of many different languages each having different input/output characteristics, generally does not support predefined logical file names. One exception on HP-UX is `/dev/null`, which is the "NULL" device or bit bucket. Other exceptions are `/dev/tty` and `/dev/umem`.

HP-UX FORTRAN, on the other hand, uses a concept of preconnected files for common input/output tasks. There is a rough correspondence between the predefined logical file names on VMS and the preconnected files available on HP-UX that you should consider.

HP-UX and other implementations of UNIX have a vastly different view of files from VMS. It is beyond the scope of this manual to discuss these differences; you should review the *HP-UX Reference* Section 9 and the *Shells: User's Guide* (the Bourne command interpreter section) to get an overview of file concepts. Only topics of concern with the preconnected files are discussed here.

Three files of special interest on HP-UX FORTRAN are standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`). By default `stdin` is the input device normally associated with your keyboard. By default, `stdout` is connected to your output device (CRT). `stderr` is similar to `stdout` except that it is normally used to report error messages rather than normal output. Unlike `stdout`, however, `stderr` is normally unbuffered so that in the event of an unanticipated halt of a program, error messages will be printed. It is normally associated with the same output device as `stdout`. All three of these files can be easily redirected from or to other files or pipes by means of HP-UX shell commands.

ANSI requires that all files be opened before they are accessed. As an extension to the standard, HP-UX FORTRAN compilers allow auto-opening of files as does VMS. You can read or write to a file that has not been opened with the `OPEN` statement. See the FORTRAN documentation for your system for details on the name of the connected file. As a convenience to you, HP-UX FORTRAN opens files automatically by associating unit 5 with `stdin`, unit 6 with `stdout` and unit 7 with `stderr`. Thus, for example, the following program will execute correctly on HP-UX.

```
      program iotest
C Note that no files have been opened by the program itself.
      write(6,100)
  100 format(' Hello world')
C PRINT statement output goes to stdout.
      print *,'HP-UX'
      end
```

Closing the preconnected files `stdin`, `stdout`, or `stderr` has no effect. However, it is allowable to reopen units 5, 6, or 7 to other files if you desire. If so, the preconnections are closed in accordance with ANSI.

In the following example, unit 6 is used for `stdout` and a user-defined file.

```
program redirect6
open (6,file='fred')
write(6,*) 'file call to file fred'
close(6)
write(6,*) 'file call to stdout'
end
```

The output to file `fred` in the current directory is

```
file call to file fred
```

The output to `stdout` (normally your CRT) is

```
file call to stdout
```

Table 6-5 shows the rough correspondence with VMS predefined logical file names.

Table 6-5. VMS Predefined File Names

VMS	HP-UX
`SYS$COMMAND`	`stdin`
`SYS$DISK`	(no default correspondence)
`SYS$ERROR`	`stderr`
`SYS$INPUT`	`stdin`
`SYS$NODE`	(no default correspondence)
`SYS$OUTPUT`	`stdout`
`SYS$LOGIN`	(no default correspondence)
`SYS$SCRATCH`	(no default correspondence) On Series 700/800, `/usr/tmp` is used for scratch files. On Series 300/400, current directory or the value of **`$TMPDIR`** is used unless an absolute path name is included.

Calling Other Languages

Most of the comments made in Chapter 5 about C calls to other languages also apply to FORTRAN except that FORTRAN frequently needs to call system routines which are written in C (see Table 6-6).

Table 6-6. FORTRAN Interfacing Compatibility

FORTRAN	Pascal	C
`CHARACTER`	`char`	`unsigned char`
`Hollerith` (synonymous with `CHARACTER`)[1]		
`BYTE`, `LOGICAL*1`[1,2]	-128..127 or `boolean`	`char`
`LOGICAL*2`[1,2]	-32768..32767 on Series 300/400; `bit16` on Series 700/800	`short`
`INTEGER*2`[1,2]	-32768..32767 on Series 300/400; `shortint` on Series 700/800.	`short`
`LOGICAL,LOGICAL*4`[2]	`integer`	`long` or `int`
`INTEGER,INTEGER*4`[2]	`integer`	`long` or `int`
`REAL,REAL*4`[2,3]	`real`	`float`
`DOUBLE PRECISION`, `REAL*8`[2]	`longreal`	`double`
`REAL*16`[1,2]	none	`long double`
`COMPLEX*8`[2]	`record`	`struct`
`DOUBLE COMPLEX`, `COMPLEX*16`[1,2]	`record`	`struct`
`RECORD`[1]	`record`	`struct`

1 Extension to ANSI 77 standard.

2 The length descriptor, $*n$, is not ANSI 77 standard.

3 Pointer variables are of type `INTEGER*4`.

You can create arrays for any of the primary data types.

In addition to the basic types, many programs must communicate with C "strings". These are emulated in FORTRAN as an array of characters the last element of which has value 0 (`CHAR(0)`). Note that HP Pascal "strings" (as opposed to packed arrays of characters) can be simulated also by an array of characters, but the characters will be offset in the array due to the length field at the front (refer to the *HP Pascal Reference* for details). When communication with FORTRAN is desired, you may want to use Pascal packed arrays of characters rather than strings.

Although the syntax for FORTRAN `RECORD`s differs from C `struct`s, default packing and alignment rules are similar between the two languages.

Another problem in interfacing FORTRAN with C or Pascal can occur because FORTRAN uses a column-major storage representation for its multi-dimensional arrays. C and Pascal use a row-major ordering. Thus for proper accessing, the order of the subscripts must be reversed (in both the declaration and usage—thus, we end up with the transpose of a matrix).

The FORTRAN `$OPTION SHORT` directive instructs the compiler to use `INTEGER*2` and `LOGICAL*2` as the defaults (when `*`*n* is not specified). This can cause communication problems when two subroutines both specify `INTEGER`, but one has this option enabled. To get around this problem, explicitly declare the length at all times. But keep in mind that doing so is non-standard, as is using the `$OPTION SHORT` directive.

Calling C

Since all the HP-UX system calls and subroutines are accessed as C functions, you may want to call a C function from a FORTRAN program. There are some basic obstacles to doing so. The major problem is that C and FORTRAN pass parameters differently—C *by value* and FORTRAN *by reference*. You can use the `$ALIAS` directive to change FORTRAN's parameter passing mechanism or the name of the external C routine as searched for by the linker `ld`. The `$ALIAS` directive is supported on all HP-UX FORTRAN implementations. Here is an example of its use:

```
PROGRAM TESTALARM
$ALIAS IALARM = 'alarm'(%val)

C set a 10 minute alarm
       I = IALARM(60*10)
C reset alarms, get time remaining on last alarm
       I = IALARM(0)
C allow any possible non-zero "time remaining" seconds count
       IF ((I .LT. 1) .OR. (I .GT. 600)) STOP 'TESTALARM FAILED'
       STOP 'TESTALARM passed'
       END
```

Series 700/800 provides a special library of FORTRAN routines that interface to HP-UX system calls. The library is named `/usr/lib/libcl.a` and is linked automatically when you compile a FORTRAN program.

Note these items:

Logicals	C uses integers for logical types. A FORTRAN 2-byte `LOGICAL` is equivalent to a C `short`; a 4-byte `LOGICAL` is equivalent to a `long` or `int`. In C and FORTRAN, zero is false and any non-zero value is true.
Files	File units and pointers can be passed from FORTRAN to C via the `FNUM` and `FSTREAM` intrinsics. A file created by a program written in either language can be used by a program of the other language if the file is declared and opened in the latter program.
Characters	Without the use of the `$ALIAS` directive, passing character data from FORTRAN to C is tricky because these languages represent character strings in completely different ways. By specifying `%ref` as a parameter passing descriptor, however, the compiler is directed to use *pass by reference* addressing, which is equivalent to passing the address of the beginning of the character variable. Within C, this is understood to be a `char` pointer. Remember that FORTRAN character strings, by default, do not contain a terminating `NUL` character as in C.

The technique shown in the following example works on all HP-UX systems. However, some other FORTRAN 77 compilers may not understand aliasing.

The example shows passing a character string from a FORTRAN program to a C function. The function returns the number of characters in the string before a space. Otherwise, it returns the maximum string length.

```
/* C program */
#define MSLEN 300

sizer(x) char *x;
{
     register int i;

     for (i=0; i <MSLEN; i++)
          if (x[i] == ' ') return(i);
     return(MSLEN);
}
```

```
C fortran program
$alias sizer='sizer'(%ref)
      program test
      character*300 x
      integer sizer
      external sizer
      integer i
      data x/"abcdefghi klmnop"/

      i = sizer(x)
      print *,i
      end
```

6

The commands to compile and link these two files are:

```
cc -c chcount.c
f77 main.f chcount.o
```

The resulting object file would be left in `a.out`.

It is possible to mix C and FORTRAN I/O via the FORTRAN **FNUM** and **FSTREAM** intrinsics. **FSTREAM** returns the C **FILE*** stream pointer corresponding to a FORTRAN I/O unit. **FNUM** returns the system file descriptor for an I/O unit. Here is an example:

```
      PROGRAM FNUM_TEST
$ALIAS IWRITE='write' (%val ,%ref ,%val)
      CHARACTER*1 A(10)
      INTEGER I, STATUS

      DO 10 J=1,10
       A(J)="X"
10    CONTINUE
      OPEN(1,FILE='file1',STATUS='UNKNOWN')
      I=FNUM(1)
      STATUS=IWRITE(I,A,10)
      CLOSE (1, STATUS = 'KEEP')

      OPEN (1,FILE='file1', STATUS='UNKNOWN')
      READ (1,4) (A(J), J=1,10)
4     FORMAT (10A1)
      DO 12 J=1,10
       IF (A(J) .NE. 'X') STOP 'FNUM_TEST FAILED'
12    CONTINUE
      IF (STATUS .EQ. 10) STOP 'FNUM_TEST passed'
      END
```

Another area for potential problems is passing arrays to C subprograms. An important difference between FORTRAN and C is that FORTRAN stores arrays in column-major order whereas C stores them in row-major order (like Pascal). Refer to "Calling FORTRAN" in Chapter 5 "Porting C Programs" for examples.

Calling Pascal

Listed below are some of the differences between Pascal and FORTRAN that may cause problems when calling Pascal from FORTRAN.

Logicals

FORTRAN has the LOGICAL type for representing boolean values. On Series 300/400, FORTRAN and Pascal share a common definition of true and false: zero is false, and any non-zero value is true. A FORTRAN LOGICAL is the same as an unpacked Pascal BOOLEAN. The FORTRAN LOGICAL*1 and LOGICAL*4 types do not have an equivalent in Pascal.

Arrays

FORTRAN stores arrays in column-major order; Pascal stores them in row-major order.

Files

A FORTRAN unit cannot be passed to a Pascal routine to perform input/output on the associated file. Nor can a Pascal file variable be used by a FORTRAN routine. However, a file created by either language can be used by the other if the file is opened and accessed by the method appropriate to that language.

Of course, files can be accessed from either language through the use of HP-UX input/output subroutines and intrinsics. (For more information, refer to the appropriate system reference manual.)

Parameter Passing Methods

By default, FORTRAN passes parameters by reference. Therefore, all parameters in a Pascal routine called from FORTRAN and all those in the external declaration of a FORTRAN routine called from Pascal must be VAR parameters. If necessary, you can force FORTRAN to pass by value with the %VAL built-in function or with the $ALIAS directive.

Complex Numbers

Pascal has no COMPLEX numbers. However, they can be represented in Pascal by the following record structure:

```
TYPE complex : RECORD
                  real_part,
                  imaginary_part: REAL
               END;
```

Similarly, a FORTRAN DOUBLE COMPLEX number can be represented by the above record structure with the real and imaginary parts being of Pascal type LONGREAL.

Hollerith and Character

The FORTRAN Hollerith and character data types are equivalent to the Pascal `PACKED ARRAY OF CHAR.`

Passing String Parameters

Pascal has several different ways of passing strings, none of which is compatible with Series 700/800 FORTRAN.

7

Porting Pascal Programs

HP-UX systems support a version of Pascal known as Hewlett-Packard Standard Pascal (HP Pascal). HP Pascal is a superset of ANSI Pascal, and it implements many advanced features. A few of the features differ between the Series 300/400 and 700/800. There are also differences between HP Pascal and the Pascal Workstation; one notable difference is calling conventions. This chapter describes:

- general portability considerations to keep in mind when porting HP Pascal programs
- porting between HP Pascal and the Pascal Workstation
- porting between HP Pascal and VMS Pascal
- calling other languages

General Portability Considerations

If you plan only to run your programs on HP computers, it will not take much work to port them and the extra features will make your programming much easier. However, if you port those programs to another vendor's computer, the effort to do so will be proportional to the use of nonstandard HP Pascal extensions. Even if the system you are moving the programs to has extensions, it is doubtful they have the same form as HP Pascal's extensions.

To help you determine which features are nonstandard in an HP Pascal source file, HP Pascal provides the `$ANSI ON` directive. To use it, simply include the following line at the start of the source file:

```
    $ANSI ON$
```

When you compile the source file using the `-L` compile line option, the Pascal compiler generates a listing file that shows where nonstandard features are used.

Using the `$ANSI ON` directive and the `-L` option helps ensure that you use only ANSI Standard Pascal features or that you will at least be aware of where you are using nonstandard features.

The rest of this section summarizes some of the HP Pascal language features, both standard and nonstandard, that may cause problems when porting HP Pascal to or from other systems.

Data Type Sizes and Alignments

Table 7-1 shows the sizes and alignments of the Pascal data types on HP-UX architectures. Note that packing significantly affects data type aligments and sizes. For more specific information, see the appropriate language reference or the *HP Pascal/HP-UX Programmer's Guide.*

Table 7-1. HP Pascal Data Types

Type	Size (bytes)	Alignment (300/400)	Alignment (700/800)
`char`	1	1-byte	1-byte
`boolean`	1	1-byte	1-byte
`shortint`	2	Not supported	2-byte
subrange of `integer` ≥ -32768 AND ≤ 32767	2[1]	2-byte	2-byte
subrange of `integer` < -32768 OR > 32767	4	4-byte	4-byte
`integer`	4	4-byte	4-byte
`longint`	8	Not supported	4-byte
`enumeration`	2[1]	2-byte	2-byte or 4-byte, based on declared range
subrange of `enumeration`	2[1]	same as host enumeration type	2-byte or 4-byte, based on declared range
`real`	4	4-byte	4-byte
`longreal`	8	4-byte	8-byte
pointer	4	4-byte	4-byte
sort (32 bit)	4	Not supported	4
sort (16 bit)	2	Not supported	2
bit52	8	Not supported	2
`$exthaddr$` pointer	8	Not supported	2
`set`	Varies	Varies	Varies

1 On Series 700/800, 1, 2, or 4 bytes can be allocated, depending on the declared range.

If the `+A` compile line option is specified, then any data types larger than two bytes are aligned on a 2-byte boundary on Series 300/400 computers.

Control Constructs

- The `TRY/RECOVER` construct is supported on all HP-UX implementations. Escape codes for errors differ significantly between the implementations.
- The `MARK/RELEASE` procedures are supported on all HP-UX implementations. There are minor differences in behavior but code is essentially portable.

Input/Output

- Series 300/400 and 700/800 differ in the way they allow association with an HP-UX file descriptor in the **`reset()`** procedure. The association is *not* similar in the **`associate()`** procedure.
- Series 700/800 uses the **`options string`** parameter on **`reset()`**, **`rewrite()`**, **`open()`**, and **`append()`** procedures. Series 300/400 ignores this parameter.
- By default, **`stdout`** is buffered on the Series 700/800. It can be changed to unbuffered via an option.
- Series 300/400 requires declaration of **`stderr`** after declaring it as a program parameter; Series 700/800 does not.
- Series 700/800 implements the **`fnum`** function; Series 300/400 does not.
- Series 300/400 and 700/800 differ in how they handle **`eof`**, **`get`**, and **`put`** with direct access files.
- The `close(`*fileVar*`)` procedure has different default behavior on each system.

Modules

- Modules are supported on all HP-UX implementations but some syntactic and semantic differences exist. For example, Series 700/800 requires that `CONST`, `TYPE`, and `VAR` declarations precede routine declarations within the `EXPORT` section whereas Series 300/400 permits them to be intermixed.
- Series 300/400 permits separate compilation only within modules. Series 700/800 can compile outside modules with the use of **`$SUBPROGRAM$`**, **`$GLOBAL$`**, and **`$EXTERNAL$`**.

Assignment to Procedure Variables

Assignment to a procedure variable has a different syntax on each of the two architectures.

Maximum String Size

On the Series 300/400, the maximum string size is 255 characters by default, but by specifying `$LONGSTRINGS$`, it can be virtually unlimited. Series 700/800 strings are essentially unlimited.

Packed Arrays and anyvar Parameters

On the Series 300/400, elements of packed arrays can be passed as `anyvar` parameters only if the `ALLOW_PACKED` compiler option has been used.

Structured Constants

All HP-UX implementations support structured constants but different restrictions may apply. Series 300/400 restricts their use to within the `CONST` section and it does not do full type checking on variant record structured constants.

longreal Precision

A small difference in precision exists between the implementations of `longreal`.

globalanyptr and localanyptr

Only Series 700/800 implements `globalanyptr` and `localanyptr`. All HP-UX implementations have `anyptr`, although minor differences exist.

anyvar Value Checking

`anyvar` is supported on all HP-UX implementations. Series 300/400 does not perform any checks to see if `anyvar` values are legitimate. Series 700/800 passes size information with `anyvar` parameters.

Miscellaneous

- Series 700/800 supports **readonly** parameters; Series 300/400 does not.
- Series 300/400 allows using a file variable as a parameter to the **sizeof** function; Series 700/800 does not.
- Series 300/400 allows you to get the address of a constant with the **addr** function; Series 700/800 does not.
- Only Series 700/800 supports **crunched** arrays and records.
- Series 700/800 does not fully support `packed array[0..`*anything*`] of char`.
- Slight semantic differences exist between Series 300/400 and 700/800 program parameters.
- Series 700/800 has the following built in functions not available on Series 300/400:

```
susizeof
statement_number
haveextension
haveoptvarparam
```

- The **assert** procedure is defined on Series 700/800 but not on Series 300/400.
- Series 700/800 requires `$STANDARD_LEVEL 'HP_MODCAL'$` before importing an argument.
- Series 700/800 does not allow a label on the statement following a **recover** statement.
- Series 700/800 allows **lobound** subrange expressions to start with "(".
- Series 700/800 scans source that has been conditionally compiled out. This allows NLS characters in conditionally compiled sections of the source.
- To convert a pointer to an integer with `ord(`*pointerType*`)`, you must compile with `$STANDARD_LEVEL 'EXT_MODCAL'$` on Series 700/800.
- Arguments for the `+` operator with strings differ on the two implementations. For instance, `chr()` cannot be used with `+` on Series 700/800.

- On Series 300/400, `waddress` does not accept NIL or a NIL-valued pointer.
- For some operations, `packed array of char` does not require a lower bound of 1 on Series 300/400.
- The two implementations generate different listings.
- Series 300/400 evaluates a procedure alias before `addr(`*alias*`)` is performed; Series 700/800 does not.

Compile Line Options

Table 7-2 summarizes the differences in compile line options between Series 300/400 and Series 700/800. See *pc*(1) for details.

Table 7-2. Differences in Pascal Compiler Command Lines

Option	Effect	Difference
`+A`	use 2-byte alignment rules	Series 300/400 only
`+bfpa`	Affects floating-point operations	Series 300/400 only
`+C`	convert MPE file names to HP-UX names	Series 700/800 only
`+DA1.0`	Optimize for Series 800 architecture and instruction set. Or, use `DA8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only
`+DA1.1`	Optimize for Series 700 architecture and instruction set. Or, use `DA7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only
`+DS1.0`	Optimize for Series 800 instruction scheduling. Or, use `DS8`*xx* where `8`*xx* is a Series 800 system model number.	Series 700/800 only
`+DS1.1`	Optimize for Series 700 instruction scheduling. Or, use `DS7`*xx* where `7`*xx* is a Series 700 system model number.	Series 700/800 only
`+ffpa`	Affects floating-point operations	Series 300/400 only
`+l`	Allows production of dynamic load libraries	Series 300/400 only
`-L`	produce a program listing	Series 300/400 goes to a file specified by **`$LIST filename$`**

Table 7-2.
Differences in Pascal Compiler Command Lines (continued)

Option	Effect	Difference
`+M`	library calls for floating point	Series 300/400 only
`+N`	turn off notes	Series 700/800 only
`+O`	optimization	Series 700/800 only
`-O`	optimize	Series 700/800 only
`+S`	use 4-byte alignment rules	Series 300/400 only
`-S`	produce assembly output	Series 700/800 only
`-T`	same as **`$TABLES ON$`**	Series 300/400 only
`+U`	same as **`$ALLOW_PACKED ON$`**	Series 300/400 only
`-y`	prepare object for static analysis	Series 700/800 only
`+z` and `+Z`	produce PIC object for shared libraries	Series 700/800 only

Inline Compiler Options (Directives)

HP-UX Pascal compilers support different (although intersecting) sets of compiler options. Additionally, some common options have different semantics, and a slightly different syntax. For portable code, keep compiler options to a minimum. Especially avoid ones that affect the semantics of the language or enable system level programming extensions, like **`$SYSPROG$`** on the Series 300/400.

The following items show options that have semantic differences on one or more of the HP-UX implementations:

`ALIGNMENT`	Series 700/800 only. Changes storage alignment for types other than strings and file types.
`ALLOW_PACKED`	Series 300/400 only. Allows `ANYVAR` parameter passing of fields in packed records and arrays, and `SIZEOF` using packed fields and arrays.

ANSI	Available on all HP-UX implementations. Series 300/400 requires that it be at the top of the file.
ASSERT_HALT	Series 700/800 only. Causes the program to halt if the assert function fails.
ASSUME	Series 700/800 only. Sets optimizer assumptions.
BUILDINT	Series 700/800 only. Causes the compiler to build an intrinsic file rather than an object code file.
CHECK_ACTUAL_PARM	Series 700/800 only. Sets level of type checking of actual parameters for separately compiled functions or procedures.
CHECK_FORMAL_PARM	Series 700/800 only. Sets level of type checking of formal parameters for separately compiled functions or procedures.
CODE	Available on all HP-UX implementations. Selects whether a code file is generated. Series 300/400 disallows this directive within a procedure body.
CODE_OFFSETS	Available on all HP-UX implementations. Causes PC offsets to be included in the listing. Series 300/400 disallows this directive within a procedure body.
CONVERT_MPE_NAME	Series 700/800 only. Same effect as +C option.
COPYRIGHT	Series 700/800 only. Causes a copyright string to be placed into object code.
COPYRIGHT_DATE	Series 700/800 only. Sets the copyright year and causes it and the copyright string to be placed into object code.
DEBUG	Series 300/400 only. Causes line number debugging information to be included in the object code.
EXTERNAL	Series 700/800 only. Used in conjunction with the GLOBAL option, enables you to compile one program as two or more compilation units.

EXTNADDR	Series 700/800 only. Specifies long pointer accessing.
FLOAT_HDW	Series 300/400 only. Controls generation of code for floating-point hardware.
GLOBAL	Series 700/800 only. Used in conjunction with the EXTERNAL option, GLOBAL enables you to compile one program as two or more compilation units.
GPROF	Series 700/800 only. Generates code for profiling.
HEAP_COMPACT	Series 700/800 only. When this and HEAP_DISPOSE are on, free space in the heap is concatenated.
HEAP_DISPOSE	Series 700/800 only. Disposed space in the heap is freed for new uses by new.
HP_DESTINATION 'ARCHITECTURE	Series 700/800 only. Generates object code for a particular version of of the PA-RISC architecture.
HP_DESTINATION 'SCHEDULER	Series 700/800 only. Performs instruction scheduling tuned for a particular implementation of the PA-RISC architecture.
IF, ELSE, ENDIF	Available on all HP-UX implementations. Controls conditional compilation. Essentially, the semantics are the same, but each implementation has minor differences in semantics. Refer to the appropriate language reference for details.
INLINE	Series 700/800 only. Causes a procedure call to be replaced by inline code.
KEEPASMB	Series 700/800 only. Causes the compiler to preserve an assembly file for the source file.
LINENUM	Series 300/400 only. Sets listing line number.
LINES	Available on all HP-UX implementations. Specifies number of lines per page on a listing. Default values are 60 for Series 300/400 and 59 for Series 700/800.

LIST_CODE	Series 700/800 only. When LIST is also on, a mnemonic listing of object code is produced.
LISTINTR	Series 700/800 only. List an intrinsic file to a specified file.
LITERAL_ALIAS	Series 700/800 only. Changes the semantics for the ALIAS option.
LOCALITY	Series 700/800 only. Causes a locality name to be written to the object file for performance enhancement.
LONGSTRINGS	Series 300/400 only. Extends the maximum length of strings from 255 characters to virtually any length.
MLIBRARY	Series 700/800 only. Specifies alternate file into which the module export text is to be written.
NOTES	Series 700/800 only. Causes helpful compiler notes to be printed on the program listing.
OPTIMIZE	Series 700/800 only. Sets level of optimization.
OS	Series 700/800 only. Specifies the run-time operating system under which this program is to be run.
RANGE	Available on all HP-UX implementations. Minor differences exist between the implementations on what items are checked. Refer to the appropriate language reference for details.
S300_EXTNAMES	Series 700/800 only. Changes external names to a form consistent with Series 300/400 conventions.
SAVE_CONST	Series 300/400 only. Controls scope of structured constants.
SEARCH	Available on all HP-UX implementations. Series 700/800 has two ways to create the list of files (one of which is the same as Series 700/800, the other uses MLIBRARY).

SEARCH_SIZE	Series 300/400 only. Changes number of external files that can be searched. The default is 9.
SHLIB_CODE	Series 700/800 only. Generates PIC object code that you can use to create libraries.
SKIP_TEXT	Series 700/800 only. Causes the compiler to ignore source code.
STANDARD_LEVEL	This is implemented on all HP-UX implementations. Series 700/800 allows the EXT_MODCAL extension level beyond HP_MODCAL.
STATEMENT_NUMBER	Series 700/800 only. When enabled, the compiler generates a special instruction to identify a code sequence with its corresponding Pascal statement.
SUBPROGRAM	Series 700/800 only. Separate compilation facility. Use modules instead on Series 300/400.
SYMDEBUG	Series 700/800 only. Emits debugger information for xdb.
SYSINTR	Series 700/800 only. Specifies the intrinsic file to be searched for information on intrinsic procedures and functions.
TABLES	Available on all HP-UX implementations. Series 300/400 forbids its use within a procedure body, whereas Series 700/800 permits it anywhere.
TITLE	Series 700/800 only. Specifies the title to appear on the program listing.
UNDERSCORE	Series 300/400 only. Causes ALIAS parameters to have an underscore added as a prefix.
UPPERCASE	Series 700/800 only. All external names are shifted to uppercase ASCII.
VERSION	Series 700/800 only. Specifies a version stamp to be placed in the object file.

XREF	Available on all HP-UX implementations. On Series 300/400, it provides an improper subset of information from the Series 700/800.

Porting between HP-UX Pascal and the Pascal Workstation

This section will be helpful if you need to port programs between the Series 300/400 Pascal Workstation and Series 300/400 HP-UX Pascal. It focuses on conversions of Pascal programs, but also comments on assembly language translation. Some of the information may not apply to Series 700/800 porting. The material presented deals with commonly encountered porting problems.

Because the Series 300/400 HP-UX Pascal compiler was developed from the Series 200/300 HP Pascal Workstation, the two implementations are very similar. However, some differences still exist when porting between the two systems. If your programs to be ported use operating system-dependent features like low-level I/O functions, then you may have a significant porting job.

Note The following does not cover the few differences between Series 200/300 and Series 300/400 Pascal Workstations. The differences that exist are documented in the Pascal Workstation documentation set.

Module Names

Module names on HP-UX Pascal can be up to 12 characters, while on the Pascal Workstation they can be up to 15.

Real Data Type

Real variables are 32 bits on HP-UX Pascal and 64 bits on the Pascal Workstation. `longreals` are 64 bits on both implementations.

Input

Although standard Pascal specifies unbuffered input on the HP-UX Pascal implementation, on the Pascal Workstation input is buffered by default. To override this, add the following statement to the beginning of your program:

```
rewrite(input,'','unbuffered');
```

lastpos

`lastpos` is not implemented on the Pascal Workstation.

linepos

`linepos` is not implemented on the Pascal Workstation.

Absolute Addressing

Absolute addressing of variables, available through `$SYSPROG$`, have little meaning on a system which uses virtual memory. Instead, the user will need to use system names. For example, to simulate the Pascal Workstation function `IORESULT`, the user may declare:

```
VAR
   ioresult['asm_ioresult']:  integer;
```

This declaration gives the user access to the `ioresult` variable. Note, however, that the above declaration also gives the user a compiler warning namely `symbol already declared` regarding `asm_ioresult`.

Accessing absolute addresses (such as on the Model 236 graphics display) will result in a system error namely `segmentation violation`. To gain access to such memory, the user must follow the techniques described in the *HP-UX Reference* under Section 7: *graphics(7)*.

File Naming

File names in programs on the Pascal Workstation are of the form:

```
VOL:FILENAME
```

With HP-UX Pascal, file naming must follow the HP-UX path naming conventions. This occurs in `$INCLUDE$`, `$SEARCH$`, `RESET`, `REWRITE`, `OPEN`, and `APPEND` statements. Since a user may execute a program from any directory, it is safest to use full path names, rather than relative paths. The following special Pascal Workstation names should translate as follows :

- `CONSOLE`: Should use the predefined file variable `output` or the name `/dev/tty` in a `rewrite` statement.
- `PRINTER`: Should use `/dev/rlp` (`/dev/lp` is usually locked from user access). Note that this bypasses the spooler, and could intermix with someone else's output.
- `SYSTERM`: Simulating this capability first requires a system call to turn off echoing, and then the statement `reset(input,'','unbuffered')`.

$SEARCH$ File Names

`$SEARCH$` file names on the Pascal Workstation must refer to either simple relocatable (`.o`) or archived (`.a`) format object files. Libraries will be maintained by the archiver (`ar`), and the compiler will need a directory in the archive file. This is accomplished by running the program `ar -ts` on the archive which creates an entry (in the archive file). This entry can be used (by the compiler and loader) to randomly access the entry points stored in the library.

Terminal I/O

Pascal on the Pascal Workstation is defined to have unbuffered terminal I/O. However, the HP-UX system buffers input based on a "line" (a string of characters, terminated by a newline). To overcome this system buffering of input into lines, the user must specify :

```
rewrite(input,'','unbuffered');
```

Heap Management

The Pascal Workstation gives you two choices for dynamic memory management. The normal mode uses MARK/RELEASE to form a simple scheme. For more general cases, $HEAP_DISPOSE$ is needed, which will then allow the DISPOSE statement to return memory to the system.

Using HP-UX Pascal, the user has three choices of memory managers: HEAP1, HEAP2, or MALLOC. HEAP1 and HEAP2 are Pascal memory managers, while MALLOC is the system library (C) memory manager.

HEAP1 provides for a simple scheme where DISPOSE returns memory to the Pascal free list, while a RELEASE returns everything above the memory pointer to the HP-UX memory system. This memory then becomes available to any other heap manager. However, this version does not allow any RELEASE to be done after any calls to MALLOC. This does not sound like much of a restriction, but consider that any system calls that you make that need memory are likely to get them via MALLOC.

HEAP2 is more flexible, and allows for coexistence with MALLOC calls. This is accomplished at the cost of additional overhead in both space (8 extra bytes are allocated forward and backward pointers) and time (a RELEASE must traverse the linked list disposing of each block).

The last scheme uses calls to the system library procedure MALLOC to allocate memory. This is a "do-it-yourself" memory allocation scheme, and it requires using $sysprog$ and anyptrs. However, since this method uses MALLOC, it is compatible with allocation by system intrinsics and C.

The HP-UX IOCTL System Call

The following program shows how to use the HP-UX system call IOCTL to modify terminal characteristics. It does unbuffered, non-echoed terminal input. IOCTL turns off echoing, sets the minimum length line to 1 character, and sets the line timeout to 0.1 seconds.

```
$sysprog$
program termtest(input,output);

{ control code constants for the IOCTL intrinsic }
const O_RDONLY = 0;
      TCGETA   = 21505;
      TCSETAF  = 21508;

type
     {simulate a C unsigned short int for bit manipulations}
     unsigned_short = packed array[0..15] of boolean;

     {simulate a C string}
     cstring = packed array[1..81] of char;

     {simulate the C struct "termio" from /usr/include/termio.h}
     termio = packed record
                c_iflag : unsigned_short;
                c_oflag : unsigned_short;
                c_cflag : unsigned_short;
                c_lflag : unsigned_short;
{                 c_line  : char;          c_line==c_cc[-1]   }
                         { note that C packs this struct tighter
                           than Pascal can.  Thus we will include
                           the c_line field as part of the c_cc
                           array }
                c_cc    : array[-1..7] of char;
              end;

var fildes,result       : integer;
    old_state,new_state : termio;
    device,buffer       : cstring;

{here are the EXTERNAL/$ALIAS definitions for the system intrinsics}

function $alias '_open'$ openx( var path : cstring ;
                                    flag : integer ) : integer;
         external;

function $alias '_read'$ readx(     fildes : integer ;
                                var buffer : cstring ;
                                    num    : integer ) : integer;
         external;

procedure $alias '_ioctl'$ ioctl(     fildes   : integer ;
                                      control  : integer ;
                                  var terminfo : termio );
          external;
```

```
begin
  device:='/dev/tty '+chr(0);
  fildes:=openx(device,O_RDONLY);
     { get the current terminal setup}
  ioctl(fildes,TCGETA,old_state);
  new_state:=old_state;
     { set the minimum number of chars for a read to 1 }
  new_state.c_cc[4]:=chr(1);
     { set the timeout after the first char to .1 seconds }
  new_state.c_cc[5]:=chr(1);
     { turn off echoing }
  new_state.c_lflag[12]:=false;
     { turn off canonical input (i.e. erase, kill, etc.) }
  new_state.c_lflag[14]:=false;
     { load this "new" terminal setup }
  ioctl(fildes,TCSETAF,new_state);
  prompt('enter your name : ');
  repeat
       { now read a single character }
    result:=readx(fildes,buffer,1);
       { now echo the successor of the char }
    if buffer[0]=chr(255) then write(chr(0))
                          else write(succ(buffer[0]));
     { stop on ^D }
  until buffer[0]=chr(4);
  ioctl(fildes,TCSETAF,old_state);
end.
```

7

Library Differences

The Pascal Workstation and HP-UX Pascal use different libraries. This manual will not discuss the differences in detail; for such information, refer to the manuals containing the information on the libraries.

For Pascal Workstation library information, see the *Pascal Procedure Library* manual.

For HP-UX Pascal library information, you will find relevant material contained in several HP-UX manuals:

- General information on the I/O library is documented in *Programming on HP-UX*.
- For graphics information, see the applicable graphics manuals.
- The system library is documented in Section 3 of the *HP-UX Reference*.

Pascal Workstation Libraries

On the Pascal Workstation, there are three primary libraries used by almost everyone:

- The DGL graphics library. This provides a high level (Pascal) interface to device-independent graphics. DGL on the Pascal Workstation is a functional copy of the HP 1000 FORTRAN DGL library. The interface has been changed to provide more relevant names for the procedures as well as a Pascal interface.
- The I/O library. This provides various levels of access to the I/O cards on the Series 300/400 system. These include HP-IB, GPIO, and a serial interface library.
- The INTERFACE library. This is a permanently loaded library (via `initlib`), which contains much of the operating system software (disk drivers, keyboard, etc.).

HP-UX Libraries

HP-UX libraries have similar capabilities as those on the Pascal Workstation, as described below.

The DGL Graphics Library. On HP-UX, the original HP1000 FORTRAN DGL library was ported creating these differences from the Pascal Workstation:

- The original procedure names were retained.
- Parameters are all passed by reference (`var`).
- Strings are FORTRAN character arrays (with a separate length parameter).
- Integers are 16 bit integers.

Two header files are provided. They should be included in each program needing access to DGL. The first header, `/usr/lib/graphics/pascal/pdgl1.h`, provides the type definitions needed for interfacing to DGL. This includes `int` and `string132`. The second header, `/usr/lib/graphics/pascal/pdgl2.h`, provides the declarations for all the `EXTERNAL` DGL procedures. It includes `$ALIAS$` statements for each procedure, such that the name from the Pascal Workstation can still be used.

Since all parameters are passed by reference, all constants must first be assigned to dummy variables. All integers must either be declared as `int` or assigned to a dummy `int`. Finally, Pascal strings must be assigned to variables of type `string132`. This is a `packed array [1..132] of char`, so direct assignments can be made for string literals, or the procedure `STRMOVE` can be used to convert from Pascal string variables.

STARBASE Library. Another graphics library is STARBASE. This package is intended to be an extension of the HP Graphics Peripheral Interface Standard, which is an extension of the ANSI standard Virtual Device Metafile and Virtual Device Interface. These (and thus STARBASE) form the basis of the Graphics Kernel System. This is a higher level ANSI standard (2D) graphics package.

The STARBASE library provides a high-performance interface to graphics hardware and other selected graphics peripherals. It provides support unavailable in DGL, with access to more device features. STARBASE is available on the 4.0 and subsequent releases of HP-UX.

SYSTEM Library. The SYSTEM library on HP-UX consists of a number of library files. These reside in the directories `/lib` and `/usr/lib`, as well as in the kernel itself. The capabilities provided exceed those available on the Pascal Workstation in many cases, but in others, they fall short. Two sections of the *HP-UX Reference* describe these capabilities in concise form. Section 2 describes the system intrinsics, which are the operating system calls. Section

3 describes the system libraries, which are the libraries for C, math, standard I/O, and various specialized libraries. The *HP-UX Reference* describes these capabilities via a C language interface (due to the fact that most of them are written in C). Pascal interfacing to any of these functions is usually fairly straightforward, with the main effort involved a result of replacing the header files that are needed.

Compiler Option Differences

The compiler options available on HP-UX Series 300/400 Pascal, with the exceptions below, are a subset of the ones available on the Pascal Workstation implementation. The following options are available *only* on the Pascal Workstation.

`CALLABS`	Switches absolute jumps on and off.
`COPYRIGHT`	Includes copyright information.
`DEF`	Changes size and location of compiler's `.DEF` file.
`HEAP_DISPOSE`	Controls garbage collection.
`IOCHECK`	Controls error checking on system I/O routine calls.
`REF`	Changes size and location of compiler's `.REF` file.
`STACKCHECK`	Controls stack overflow checking.
`SWITCH_STRPOS`	Switches order of parameters for the `STRPOS` function.
`UCSD`	Allows use of UCSD Pascal extensions. UCSD extensions are not and will not be implemented on HP-UX. There are simple fixes for most of these capabilities. Most notably, the UCSD string functions are supported through Pascal string functions. Also, to allow case statements to "fall through," an `OTHERWISE` clause is needed.

In addition, the compiler option `PARTIAL_EVAL` is implemented differently on the two systems. The default on the Pascal Workstation is `OFF`, but the default on HP-UX Series 300/400 Pascal is `ON`. This was done to make HP-UX Series 300/400 Pascal compatible with previous HP-UX Pascal implementations. Note that this is different from early releases of Series 300/400 HP-UX Pascal.

Assembly Language Conversion

The conversion of assembly language routines from the Pascal Workstation to HP-UX is fairly straightforward. An HP-UX command exists on the Series 300/400 called `atrans` which translates a Pascal Workstation assembly language source file into an HP-UX assembly language source file using the assembly syntax available since release 5.15. On HP-UX, the external names are referenced via 32 bit addresses, so the code size may grow. Also, many of the assembler directives will not port directly to HP-UX, but some of the important ones have replacements.

The following are points to be aware of when converting:

- Absolute displacements off the program counter cannot be guaranteed to translate correctly. Any line referencing the program counter will be flagged by a warning message.
- The HP-UX assembler restricts expressions involving forward references for which `atrans` makes no check. Such references may involve only a single symbol, a symbol plus or minus an absolute expression, or the subtraction of two symbols.
- The character `@` is not accepted as a valid identifier character on the HP-UX assembler. It is translated to `A` and a warning is issued.
- Lines containing the following pseudo-ops have no parallel on the HP-UX assembler and are translated as comment lines: `decimal`, `end`, `llen`, `list`, `lprint`, `nolist`, `noobj`, `nosyms`, `page`, `spc`, `sprint`, and `ttl`.
- Lines containing the `mname`, `include`, and `src` pseudo-ops are translated as comment lines, and a warning is printed.
- The following pseudo-ops require manual intervention to translate: `com`, `lmode`, `org`, `rorg`, `rmode`, `smode`, and `start`. Each line containing these pseudo-ops will cause a message to be printed stating that an error will be generated by the HP-UX assembler.
- When specifying certain addressing modes, the Pascal Workstation assembler allows some operands to appear out of order, whereas the HP-UX assembler does not. `atrans` does not rearrange these into proper order.

Porting between HP Pascal and VMS Pascal

To provide some information to help evaluate the task of porting from VMS Pascal to HP Pascal, the following comparison between the two languages is included. The listing is by no means complete.

Lexical Elements

ASCII Character Set

Both languages use the same character set. HP Pascal may have extensions for Native Language Support.

Special Symbols

The following VMS Pascal symbols are not recognized by HP Pascal:

- Exponentiation (**)
- Type case operator (: :)

Reserved words

The following VMS Pascal reserved words are not recognized by HP Pascal:

```
%DESCR REM
%DICTIONARY VARYING
%IMMED VALUE
%REF
%STDESCR
```

The following HP Pascal reserved words are not recognized by VMS Pascal:

```
export
implement
import
nil
```

Directives

The following is the syntax of VMS Pascal directives:

```
%<directive> ...
```

The following is the syntax of HP Pascal directives:

```
$<directive> ...$
```

HP Pascal does not support VMS Pascal `EXTERN` or `FORTRAN` directives.

Identifiers

VMS Pascal allows a `$`; HP Pascal does not.

Predefined Identifiers:

The following VMS Pascal identifiers are not recognized by HP Pascal:

ADD_INTERLOCKED	FIND_MEMBER	SNGL
ADDRESS	FIND_NONMEMBER	STATUS
ARGUMENT	IADDRESS	STATUSV
ARGUMENT_LIST_LENGTH	INDEX	SUBSTR
BIN	INT	TIME
BITNEXT	LINELEMIT	TRUNCATE
BYTE_OFFSET	LOCATE	UAND
CARD	LOWER	UDEC
CLEAR_INTERLOCKED	MAX	UFB
CLOCK	MIN	UNDEFINED
CREATE_DIRECTORY	OCT	UNLOCK
DATE	ODD	UNOT
DELETE_FILE	PRESENT	UNIGNED
DBLE	QUAD	UOR
DELETE	QUADRUPLE	UPPER
ESTABLISH	READV	URUND
EXPO	RENAME_FILE	UTRUNC
EXTEND	RESETKREVERT	UXOR
FIND	SET_INTERLOCKED	WRITEV
FIND_FIRST_BIT_CLEAR	SINGLE	XOR
FIND_FIRST_BIT_SET	SIZE	ZERO
FINDK		

Compilation Unit Structure

Syntax for module declarations differs.

Declarations

The following differences exist:

- HP Pascal restricts types of constant expressions in constant definitions.
- HP Pascal does not support VMS Pascal attribute lists.
- HP Pascal does not support initialization of variables within the variable declaration.

Data Types

Unsigned integers are not supported in HP Pascal.

VMS PASCAL `D_floating`, `G_floating`, `double`, and `quadruple` real numbers are not supported by HP Pascal.

VARYING of CHAR is not supported in HP Pascal.

Expressions

HP Pascal does not support the same set of compile-time expressions as VMS PASCAL.

Operators

The syntax of type casts is different.

Statements

Statements are compatible.

User Declared Routines

HP Pascal allows redeclaration of forward procedure or function parameters when the routine is defined.

For example:

```
procedure p ( var arg1,arg3:integer); forward;
.
.
.
procedure p ( var arg1,arg3:integer);
var
 ...
begin
   ...
end; {procedure p}
```

Declarations of formal parameters differ. In particular, HP Pascal does not support:

- a value-section for default values for formals.
- a foreign section.
- varying conformant arrays.

Functions

For some provided functions such as `open` and `close`, the types and value of arguments differ.

Calling Other Languages

Pascal has seven basic types, along with pointers, records, and subranges. The user may also create arrays of each of these. Pascal can pass its parameters by value or by reference. Compatibility of these types with the other languages is shown in Table 7-3.

Table 7-3. Pascal Inter-Language Compatibility

Pascal	C	FORTRAN
`boolean`	`unsigned char` `unsigned`	`logical*1` (`logical*2` and `logical*4` will not work)
`char`	`char`	`character*1`
`integer`	`long; int`	`integer` (*4)
−32768..32767 (`shortint` on S700/800)	`short`	`integer` (*2)(extension to ANSI standard FORTRAN 77)
`real`	`float`	`real` (*4)
`longreal`	`double`	`double precision`
enumerated type	`enum`	use `integer*2` (extension to ANSI standard FORTRAN 77)
subrange (32 bit)	use `long; int`	use `integer*4`
subrange (16 bit); S300/400 only	use `short`	use `integer*2` (extension to ANSI standard FORTRAN 77)
`set`	none	none
`record`	`struct` (the fields must align)	`record`
^*type* (pointer)	*type* * &*var*	none
var^ (dereferencing)	**var*	none
sort (16 bit); S700/800 only	`unsigned short`	none
sort (32 bit); S700/800 only	`unsigned short`	none

Take care when using packed records in Pascal because the compiler packs the data into the smallest required space. Thus, fields may not align on byte boundaries. This makes it difficult to access the data from C and FORTRAN.

If Pascal routines are to be called from FORTRAN, make sure to declare all parameters as `VAR` parameters.

To call an external (FORTRAN, C, or assembler) procedure on Series 300/400 computers, the user must declare a Pascal interface to it, and then define it as `EXTERNAL`. Pascal will then add an underscore (_) prefix to the name; this is the name that the loader will look for. If the user wishes to use a different name (in the Pascal code), or if the routine is an assembler routine (the assembler does not have a _ prefix on its external names), then the **`$ALIAS$`** directive is needed in the interface declaration. C and FORTRAN also use a _ prefix, so names will match properly.

A similar situation exists on Series 700/800 except that the underscore is neither added nor required for external names.

Refer to the *HP Pascal/HP-UX Programmer's Guide* for more information on calling other languages.

Calling C

HP-UX system calls and subroutines are actually C functions, so if your program calls such routines, you must know how to call C functions. This section contains a list of relevant issues and some examples of calling a C function from a Pascal program.

- C does not have subroutines; it has functions that may or may not return a result. The default type of the returned value is integer, but other types may also be returned. Since the C function will not be defined in the same source file as your Pascal program, you will have to declare the C function as an external Pascal function within the source file. It is important for you to make the external declaration correspond to the definition of the C function.
- Pascal gives you the choice of passing parameters by value or by reference. C passes all parameters by value, but you can emulate passing by reference by declaring a formal parameter as a pointer. This relationship is important to understand when writing the external function declaration through which Pascal "sees" the C function. If the C function you are calling has a formal parameter declared as a pointer, then in your Pascal external declaration of the function, the formal parameter should be a `var` parameter. All C formal parameters that are not declared as pointers should have corresponding Pascal non-`var` actual parameters. See the example below for clarification.

- Records and `structs` can be easily passed between C and Pascal as long as the Pascal records are unpacked. Packed records introduce system dependent problems that are not discussed here.
- Both C and Pascal store arrays in row-major order so they may be passed successfully. When passing character arrays (which are actually pointers to chars), make sure that they are terminated with `chr(0)`. Always be sure to debug the interface between the two languages. Do not assume that it works just because the function works when called by a program in the same language.
- On Series 300/400, if you want to refer to an external function by a name other than the one it is defined under, use the `alias` directive to set the name.

This example shows how to call a user-defined C function from a Pascal program. First is shown the Pascal source:

```
{ SHORT PROGRAM TO CALL C FUNCTION }
program call_c(input,output);

const   str_length = 50;

type mystring = packed array[1..str_length] of char;

var     x : real;
        s : mystring;

{ DECLARE THE C FUNCTION AS AN
  EXTERNAL PASCAL FUNCTION }
function c_sub (var strng : mystring): real; external;

begin
   s:= 'abc';
   s[4]:= chr(0); { PUT NULL AT END }
   x:= c_sub(s);  { CALL THE FUNCTION }
   writeln(x)
end.
```

Here is the C source:

```
#include <stdio.h>
/* C FUNCTION TO PRINT A STRING
   AND RETURN A REAL VALUE. */
float c_sub(str)
        char    *str;
{
   printf("\n %s",str);
   return(1.211);
}
```

The commands for compiling and linking these two source files is:

```
cc -c c_sub.c
pc call_c.p c_sub.o
```

Executing the file named `a.out` will produce:

```
abc       1.211000E+00
```

The following page has an example that calls the HP-UX system function **`truncate`** from a Pascal program. The **`alias`** directive is used to rename the external symbol **`truncate`** to **`chop`** within the program. Note particularly the section that inserts a null (`chr(0)`) into the character array at the end of the file name. This is necessary because C expects all strings to be terminated by a null.

```
program chopfile(input,output);
{ PROGRAM TO TRUNCATE A FILE TO A GIVEN LENGTH }
const   str_length = 50;

type    mystring=packed array[1..str_length] of char;

var     fname : mystring;
        lngth, dummy, i : integer;

function $alias 'truncate'$ chop(var path : mystring;
                  length : integer); integer; external;
begin
   writeln('Enter name of file to be chopped: ');
   readln(fname);

   { PUT NULL IN FIRST SPACE }
   i:= 1;
   while (fname[i] <> ' ') do
      i:= i + 1;
   fname[i]:= chr(0);

   writeln('Enter new length: ');
   readln(lngth);

   { CALL THE SYSTEM FUNCTION
      WITH ITS ALIASED NAME }
   dummy:= chop(fname,lngth);

   if dummy <> 0 then
      writeln('CALL FAILED')
end. { CHOPFILE }
```

Use the following commands to compile and run this program:

```
pc chopfile.p
a.out
```

Calling FORTRAN

Listed below are differences between FORTRAN and Pascal that may cause problems when calling FORTRAN from Pascal. For more information on interlanguage calling conventions on Series 300/400, refer to the book *HP-UX Assembler Reference and Supporting Documents* (Series 300/400).

Booleans

FORTRAN has the `LOGICAL` type for representing boolean values. On Series 300/400, FORTRAN and Pascal share a common definition of true and false: Zero is false and any non-zero value is true. A FORTRAN `LOGICAL`, which is 2 bytes in size, is the same as an unpacked Pascal `BOOLEAN`. The FORTRAN `LOGICAL*1` and `LOGICAL*4` types do not have an equivalent in Pascal.

Arrays

Pascal stores arrays in row-major order; FORTRAN stores them in column-major order.

Files

A FORTRAN unit cannot be passed to a Pascal routine to perform input/output on the associated file. Nor can a Pascal file variable be used by a FORTRAN routine. However, a file created by either language can be used by the other if the file is opened and accessed by the method appropriate to that language.

Of course, files can be accessed from either language through the use of HP-UX input/output subroutines and intrinsics. (For more information, refer to the appropriate system reference manual.)

Parameter Passing Methods

By default, FORTRAN passes parameters by reference. Therefore, all parameters in a Pascal routine called from FORTRAN and all those in the external declaration of a FORTRAN routine called from Pascal must be `VAR` parameters. If necessary, you can force a FORTRAN function to pass by value with the `%VAL` intrinsic function or the `$ALIAS` directive.

Complex Numbers

Pascal has no COMPLEX numbers. However, they can be represented in Pascal by the following record structure:

```
TYPE complex : RECORD
                  real_part,
                  imaginary_part: REAL
               END;
```

Similarly, a FORTRAN DOUBLE COMPLEX number can be represented by the above record structure with the real and imaginary parts being of Pascal type LONGREAL.

COMPLEX*16, however, cannot be represented in Pascal because Pascal does not support 16-byte real values.

Hollerith and Character

The FORTRAN Hollerith and character data types are equivalent to the Pascal PACKED ARRAY OF CHAR.

Index

HEWLETT® PACKARD

Reorder No. or
Manual Part No.
B2355-90025

Manufacturing
Part No.
B2355-90625

B2355-90625