HP-UX Portability Guide

HP 9000 Series 300/800 Computers

HP Part Number 98794-90047

HEWLETT
PACKARD

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Legal Notices

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

September 1989 ... Edition 1. This edition reflects the changes that have made the Series 300 and Series 800 more compatible, especially in the areas of FORTRAN and system calls. Chapter 7, Accessing Series 300 Shared Memory, is no longer pertinent and has been removed. The Series 310 is no longer supported, and references to it have been removed. This book replaces the *HP-UX Portability Guide,* part number 98794-90046.

Contents

1

Introduction

This manual presents guidelines and techniques for maximizing the portability of C, Pascal, and FORTRAN programs on the HP 9000 computers with the HP-UX Operating System. This manual does not discuss portability of programs written for the Integral Personal Computer.

This manual concentrates on moving C, FORTRAN, and Pascal source code from one system to another and provides a general overview in these areas:

- `Porting existing code from operating systems other than HP-UX to the HP-UX environment.` You get information about known differences between the systems and the languages. You should understand the differences before attempting to port code to the HP-UX environment. In some cases, you get tactics to make the task easier.
- Porting C, FORTRAN or Pascal source code from one HP-UX architecture to another. The descriptions are specific to the Series 300 and 800 systems. Some of this information can be useful to people who port code from another operating system. You get descriptions of communication problems that exist between languages. The idea is to make the various languages compatible so that code can be reused rather than needing to be rewritten across language boundaries.
- Summarizes system calls and functions known to be system dependent. You are given highlights of the more important differences.

Within these three general goals, the table on the following page named "Manual Contents by Chapters" describes specific chapters. One chapter, "Porting from BSD4.3 to HP-UX", and one section, "The Pascal Language" in the chapter called "Porting VMS Code to HP-UX", have not yet been written. The material will be added at an appropriate time.

Table 1-1. Manual Contents by Chapters

Chapter Number and Name	Description of Contents
1: Introduction	Introduces the manual and discusses some general topics.
2: Porting VMS Code to HP-UX	■ Mentions general aids and FORTRAN utilities. ■ Discusses system differences related to FORTRAN, VMS, runtime library calls, graphics, windows, and C. ■ Discusses FORTRAN, C, and Pascal languages.
3: Porting from BSD4.3 to HP-UX	Not written as yet.
4: Porting Across HP-UX	Discusses porting across HP-UX versions according to requirements for the C, FORTRAN, and Pascal languages.
5: System Calls and Subroutines	Discusses HP-UX system calls and subroutines.
6: Pascal Workstation to HP-UX	Discusses the porting of Pascal to HP-UX in terms of differences in compiler options, features, and libraries. Also includes graphics and assembly-language conversion.
7: Accessing Series 300 Shared Memory	Describes how to utilize shared memory between cooperating processes (works only in Series 300 systems).

Overall, the manual provides a picture of portability as it exists now and refers to the 7.0 release of Series 300 and Series 800. When in doubt about features for C, FORTRAN or Pascal, refer to the HP-UX *C Programmer's Guide* or the FORTRAN or Pascal reference for your system.

A Philosophy of Portability

A software engineer needs to have the right attitude to develop portable software. In the process of developing software, the following things can hinder porting your code to another environment:

- Non-standard language extensions.
- Assembly code.
- Hardware dependencies.
- Absolute addressing.
- Floating point comparisons.
- Software "tricks" that exploit a particular architecture.

These things are discussed throughout the manual in relation to current topics. The constant idea is to use programming techniques that minimize, eliminate, or avoid system dependencies.

Standards

The use of industry standards is crucial to portability. Hewlett-Packard Company tracks these standards in the following ways:

- HP-UX is a licensee of UNIX™[1] System V.2. All HP-UX implementations pass SVVS validation.
- HP-UX has added selected 4.2bsd and 4.3bsd 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 HP9000 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. It is the intent to make HP-UX track this standard.
- Likewise, Hewlett Packard has announced a commitment to track the developing X/OPEN standard.

[1] UNIX is a trademark of AT&T Bell Laboratories, Inc.

Each language described in this manual is also subject to industry standards.

Standards for C

For C, no formal standard exists although the ANSI X3J11 committee has one under development. Nevertheless, the C language is well standardized across the industry. The core of the language implemented on HP-UX derives from *The C Programming Language,* by Kernigan and Ritchie. Since the publication of this book several near universal extensions have been added. All these extensions are common to all architectures on HP-UX. This manual will describe the extensions. C has the reputation of being the most portable of the three languages.

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. Series 300 and 800 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. They will be described later in the manual. HP-UX FORTRAN on all implementations also conforms to FIPS PUB 69-1 and ISO 1539-1980.

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. It is also an internal corporate standard to which Series 300 and 800 implementations conform or are converging. Pascal on these architectures conforms to ISO 7185-1983 level 0 at present.

Guidelines

The following items provide guidelines for making your code portable:

- Structured programs are easier to understand. Any program of even moderate complexity must be understood to be ported successfully. A well designed program that emphasizes modularity will be inherently more portable.

- Isolate system dependent code. "Include" files, libraries, and conditional compilation can make this task much easier. Calls to system routines not described in the relevant language standards are often system dependent, particularly when porting from a non-HP-UX environment.
- Avoid the use of language extensions if at all possible. In most cases they are a matter of convenience not necessity.
- 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 tuned in a system dependent manner. The apparently conflicting goals of portability and performance can be met by a careful design of a select number of encapsulated interface routines.
- The beginning of each chapter in the remainder of this manual sets forth some of the basic problem areas for a particular source language. Review these areas before finalizing your design.

Some General Considerations

Since programming languages define the meaning of a program, they are the primary concern of portability. Unless the semantics of a language are exactly the same on two different systems, you cannot assume that a program written in that language will produce the same results on both systems. Also, one implementation of a language may support extensions that are not available on other systems.

Compiler Directives

Compiler directives are a mixed blessing. There are directives available on HP-UX that generate warnings for non-standard language features. These are very useful and are covered under each language. On the other hand, some directives enable system dependent features that dissolve any hope of portability. In any case, the directives should be reviewed when porting because it is unlikely that the systems you are porting between support all the same directives. You must balance the current usefulness of each directive against its potential for portability problems.

Floating Point Fuzziness

Floating point operations can complicate compatibility. Computer floating-point numbers are usually only close approximations of real numbers, so when doing floating point compares, 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;
```

where `err_margin` is a constant representing the margin of error for comparisons. `Err_margin will` *not* be constant across all HP-UX implementations.

Series 300 Floating Point Options

Hardware

Here are the Series 300 hardware configurations with floating point math performance:

Series	Processor	Floating Point Coprocessor	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
350	68020	68881	Optional HP 98248A
360	68030	68882	Optional HP 98248B
370	68030	68882	Optional HP 98248B

The Motorola 68020 and 68030 processors differ in their speed and internal architecture, but use the same instruction set. The Motorola 68881 and 68882 floating point coprocessors use a compatible instruction set.

Determining Your Hardware Configuration

You can determine the hardware configuration of your Series 300 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).

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 boolean functions are defined to interrogate these flags:

```
    is_68010_present,
    is_68881_present (MC68881 and MC68882 are synonymous here)
    is_98248A_present (98248A and 98248B are synonymous here)
    is_98635A_present
```

Getcontext() exists as both a command and a system intrinsic function to return configuration information in string form. See the *HP-UX Reference* for details.

Compiler Options

Option	Series	Generation of Calls for Floating Point Math	Considerations
default	3xx	inline coprocessor instructions	
`+M`	3xx	user provided library calls if present; else, `libc.a` and `libm.a`	user provided calls usually have slower performance
`+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 3xx except 310

Recommendations

The `+bfpa` option for the 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.

When using the `+bfpa` option, around 90% of all computation time is spent within 10% of the executable code. 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 800 Floating Point

Series 800 has floating point support built into the CPU and therefore it does not require any external floating point support.

2

Porting VMS Code to HP-UX

This chapter presents guidelines that can help you port code from the VAX VMS^{TM1} system to the HP-UX system. The guidelines do not provide solutions for every problem; in many cases no satisfactory general solutions are known. However, the chapter attempts to inform you of the differences so that specific solutions can be developed.

General Portability Aids

HP-UX provides tools for C, FORTRAN and Pascal that may help discover some nonportable constructs. Typically these tools perform a static analysis of a source program to find nonstandard or dubious programming

For C, `lint` attempts to check for unreachable statements, poorly structured loops, unused variables, and inconsistent function use. It is an effective first level portability tool.

All HP-UX FORTRAN compilers have the `-a` option to warn about non-ANSI features. Unlike `lint`, the compilers will still produce object code when this option is selected. Keep in mind that when this option is used, the compiler can produce copious quantities of non-fatal warning messages so it is generally useful to redirect `stderr` to a file for more leisurely viewing.

Series 300 has an additional tool, `flint`, that provides a similar functionality for FORTRAN that `lint` provides for C. It too is a distinct program that does not generate object code.

All HP-UX Pascal compilers have the `-A` command line option to warn about non-ANSI features.

1 VAX and VMS are trademarks of Digital Equipment Corporation.

All HP-UX C, FORTRAN, and Pascal implementations provide support for including source statements from another specified file or device file. This facility can be very useful to help encapsulate system dependent source code by requiring minimum changes to source. Although `include` statement syntaxes differ by language, all HP-UX implementations within each language use the same syntax and semantics. For example, the FORTRAN statement:

```
INCLUDE 'foo'
```

will cause source statements from file `foo` to be included into the current input in the same manner in all HP-UX FORTRAN implementations.

Miscellaneous FORTRAN Utility Programs

Some utility programs have been provided for use with FORTRAN source code. Ratfor, a "rational" FORTRAN dialect preprocessor, translates a superset of FORTRAN that adds 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 wholesale rewriting of existing FORTRAN into ratfor will be to your advantage.

Another utility that will be useful is asa, a filter that interprets ASA carriage control characters. These carriage control characters will be ignored 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 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
```

If this program is compiled and executed without asa by 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.
```

System Differences

The work involved in porting an application from the DEC VAX VMS environment to HP-UX depends on how much the application uses features outside of the implementation language.

FORTRAN Application: No VMS System or Runtime Library Calls

If there are no VMS system or runtime library calls, and the application is written completely in FORTRAN, and it uses only FORTRAN I/O facilities, then the language comparison below can be consulted on the differences between the FORTRANs on these systems. In general, the 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 the reader to HP-UX. If the writer program runs on HP-UX, the HP-UX reader program will read the file correctly (of course). If the writer runs on VMS, and the data file is moved to the HP-UX over a local area network or on 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 in C 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 that depends on the byte ordering of integers prints 262150 on an HP-UX system and prints 393220 on a VAX system.

FORTRAN Applications With VMS System or Runtime Library Calls

To check for VMS system and runtime 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 runtime 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.

You can either write emulation routines in C or FORTRAN that use HP-UX system and library calls or modify the source to use HP-UX routines directly. (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.) 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. A programmer undertaking this task will need to get very familiar with sections 2 and 3 of the *HP-UX Reference* in addition to being knowledgeable about VMS and the application program.

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™ [2] (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.

[2] Tektronix is a trademark of the Tektronix Corporation.

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 but available as extensions in VMS FORTRAN. Consequently, an X application written in VMS FORTRAN will not just 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. However, X applications written in C should be highly portable.

C language applications

Pointers, bit fields, structs and unions are all available to produce system dependent (and nonportable) code that relies on byte ordering, alignment restrictions, and pointer representations. For information on how to write highly portable code, see "The C Language" section in this chapter.

Since the C language 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 problems.)

The C Language

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

1. There is a high degree of compatibility within the HP-UX family and between HP-UX C and other common industry implementations of C.
2. 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 encapsulates the thorniest issues of portability.

In most cases HP-UX C is a superset of VMS C. Therefore, porting from VMS to HP-UX is easier than porting 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. See Table 2-2 for specifics.
- Type `char` is signed by default on 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. Variables are local or global based strictly on scope or `static` class specifiers on HP-UX.

- 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.
- Bitfields 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 if 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 they 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 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 2-1. 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 2-2. 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

- VMS C permits the use of *$* within an identifier. This is not supported on HP-UX.
- 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 the `register` declaration causes an integral or pointer type to be assigned a dedicated register to the limits of the system unless full optimization is requested, in which case the compiler ignores register declarations. Series 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 be auto. HP-UX compilers will report an error.

- Type conversions on both systems follow the usual progression on implementations of UNIX. Both systems use conventions commonly referred to as *unsign preserving.* For example, a binary arithmetic operation involving an unsigned short will coerce that operand to an unsigned int before it is used in the operation and the type of the result will be unsigned. Conversions of floats or doubles to int are done by truncation on both systems.
- 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 NULs. 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 if the numerical value exceeds 127 (the overflow is silent). Series 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 implicitly 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. For example:

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

  is permitted on VMS and Series 300 but not on Series 800.

Preprocessor Features

- VMS supports an unlimited nesting of `#includes`. HP-UX guarantees only 11 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 words 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 words are predefined on HP-UX but not in VMS:

  `hp9000s200` on Series 300
  `hp9000s300` on Series 300
  `hp9000s800` on Series 800
  `hpux` and `unix` on all HP-UX systems.
- HP-UX preprocessors do not include white space in the replacement text of a macro. The VMS preprocessor includes the trailing white space. If your 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)` in the *HP-UX Reference* for a description and examples.
- `Curses` is supported on VMS and all HP-UX implementations. See `curses(3X)` in the *HP-UX Reference* 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 also as macros. HP-UX supplies them as macros and it also supplies the system 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: VAX-CURSE.OLB, SYS$LIBRARY:VAXCRTLG.OLB or SYS$LIBRARY:VAXCRTL.OLB to perform C input/output operations. The HP-UX stdio utilities are included in /lib/libc.a, 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 NUL terminated on VMS whereas it is guaranteed to be so on HP-UX.
- The commonly used HP-UX names end, edata and etext are not available on VMS. They are available on HP-UX.

The FORTRAN Language

Because VAX VMS FORTRAN has been a popular programming environment for many years, an enormous reservoir of FORTRAN programs exist. 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 that are also irksome.

Both the VMS and the HP-UX compilers support the full ANSI FORTRAN77 standard and Mil-Std-1753 extensions. However, the VAX VMS compiler has evolved from ANSI FORTRAN66, 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. Consequently, this section primarily describes the differences between the extensions to the FORTRAN77 standard.

Comparisons of Features

The next several subsections compare the features of VAX VMS extensions to ANSI FORTRAN 77 and HP-UX implementation notes for each feature. Each item is supported on all HP-UX implementations unless stated otherwise. The features change dramatically from one release to the next. This manual reflects only the Series 300 and Series 800 7.0 releases.

Character Sets

- Lower case ASCII letters are folded onto their upper case counterparts except within Hollerith or quoted strings. This is the default for VMS and HP-UX. HP-UX also provides compiler options for making lower case and upper case ASCII characters distinct.
- <tab> characters. Tabs have the same behavior on VMS and HP-UX implementations between columns 1-72.

- Quotation mark ("), underscore (_), exclamation point (!), dollar sign ($), ampersand (&), and percent sign (%) are supported.
- <Control> L within source code for newpage.
- Left and right angle brackets (< and >). These are used only to delimit variable expressions within formats. The variable must be simple on Series 800.
- Radix-50 character set is not supported on HP-UX.

Control Statements

- The DO ... WHILE control construct.
- The DO ... END DO control construct.
- Forcing FORTRAN66 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 so long as control eventually returns back to within the DO loop by means of an unconditional GOTO are allowed.

Data Types and Constant Syntaxes.

- The BYTE data type.
- The LOGICAL*1, LOGICAL*2, LOGICAL*4 data types.
- The INTEGER*2 and INTEGER*4 data types.
- The REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types.
- REAL*16 is supported on Series 800 only.
- The DOUBLE COMPLEX data type (synonym for COMPLEX*16).
- Octal constants of the form `O'ddd'` or `'ddd'O`.
- Hexadecimal constants of the form `Z'ddd'` or `'ddd'X`.
- Octal, hexadecimal and Hollerith constants are considered to be "typeless" and may be used anywhere a decimal constant may be used.

- Hollerith is supported on all HP-UX implementations but in different ways. On HP-UX compilers Hollerith is treated internally as a synonym for a quoted character constant.
- Character constants have a maximum length of 2000 characters on VMS and on Series 300, unlimited length on Series 800.
- RECORD and STRUCT data types are supported on both Series 300 and 800. Default alignment differs from VMS. On the Series 300 only, use the `NOSTANDARD ALIGNMENT` directive to force VMS alignment
- Octal constants of the form `"ddd` are not supported on HP-UX.
- REAL*8 (D_floating) and COMPLEX*16 (D_floating) are not supported on HP-UX.

General Statement Syntax and Source Program Format

- 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.
- Sequence numbering in columns 73-80. VMS ignores sequence numbers. HP-UX ignores anything in columns > 72.
- 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. Not supported on HP-UX.
- DATA statements can be interspersed with specification statements.
- DATA statements can be interspersed with executable statements only on Series 300, and the `-K` option must be specified.
- Alternate forms of data type length specification, 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. Beware that precedence may be changed; for example:

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

- INCLUDE 'Library (module)' for including selected library routines is not supported on HP-UX.

Input/Output Statements

- DECODE/ENCODE.
- NAMELIST directed I/O.
- List directed internal I/O.
- Files opened for DIRECT access can have sequential I/O operations performed.
- The TYPE statement.
- 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. This is on Series 300 only.
- Variable format expressions are supported on Series 300 and partially on 800 where the format expression can be a variable or a symbolic constant.

- The RECL I/O specifier counts words on VMS but bytes on HP-UX.
- The FILE I/O specifier must be CHARACTER on HP-UX.
- The ACCESS='APPEND' specifier for the OPEN statement is not supported on HP-UX.
- `O` and `Z` field descriptors.
- The `$` edit descriptor.
- The `H` field descriptor can be used only with WRITE on HP-UX. VMS permits it to be used with READ as well.
- The `Q` edit descriptors is supported on Series 800 only.
- Default field descriptors are not supported on HP-UX.
- `$` and `ASCII NUL` carriage control characters are not supported on HP-UX.
- The `ACCEPT` statement.
- `DEFINE` and `FIND` are not supported on HP-UX.
- `DELETE`, `REWRITE`, and `UNLOCK` statements are supported on Series 800.
- Key-field and key-of-reference specifiers are supported on Series 800.
- The VMS concept of indexed file access is supported only on Series 800.
- Unsupported VMS keywords and I/O specifiers are flagged.

- The following VMS keywords are supported only on Series 800 while Series 300 gives a nonfatal warning and ignores the keyword clause:

```
KEY
KEYED
KEYID
READONLY
RECORDTYPE
SHARED
```

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

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

- 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 the Series 300 only; for example:

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

Intrinsic Functions

- Mil-Std-1753 intrinsics ISHFT, ISHFTC, IBITS, BTEST, IBSET, and IBCLR are supported.
- The Mil-Std-1753 subroutine MVBITS is supported.
- ZEXT 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:

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	
CDABS	IIABS	IINT	IMIN0	JIDNNT	JMAX0	
CDEXP	IIAND	IIOR	IMIN1	JIEOR	JMAX1	
CDLOG	IIBITS	IISHFT	ININT	JINT	JMIN0	

- VMS “system” support subprograms (DATE, EXIT, IDATE, TIME, SECNDS, RAN) are supported on Series 300 and 800 as compiler options. These routines are not compatible with HP-UX system functions of the same name. ERRSNS is not supported on HP-UX.
- VMS specific intrinsics to support the REAL*16 data type are supported on Series 800.

Specification Statements

- IMPLICIT NONE turns off default type rules for variables.
- This limited number of the intrinsic functions is 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

  Arguments to these functions must be constants in this context.

- 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.
- Symbolic constants may themselves be used to define COMPLEX symbolic constants within a PARAMETER statement on VMS but not on HP-UX.
- The NOF77 interpretation of the EXTERNAL statement (non-ANSI semantics) is not supported on HP-UX.

Subprograms

- Entries in a subprogram must all be the same type, but may have different length specifiers.

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

- Actual parameters may be octal or hexadecimal when the corresponding formal parameters are CHARACTER type.
- Hollerith actual arguments are permitted. However, the corresponding formal parameter should be CHARACTER type.
- `%loc` is recognized as a built-in function to compute the internal address of a datum.
- `%val`, and `%ref` are supported within actual argument lists, but `%descr` is not. 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(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 only on Series 300.

Symbolic Names

- 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.
- Underscore in names.
- Dollar sign in names.

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.
- Noninteger 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.

- Logical operands can appear in arithmetic expressions and numeric operands can appear in logical expressions on Series 300. They can appear on Series 800 if the compiler directive `HP9000 LOGICALS` or `FTN3000_66 LOGICALS` is used.

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. By default on HP-UX, logical `.false.` is represented as 0 on HP-UX and `.true.` is represented by any non-zero bit pattern. This difference is noticeable chiefly when equivalencing LOGICAL variables to INTEGER variables. Series 300 and 800 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 onto 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 set to large default values and are further configurable by your system administrator. See the *HP-UX System Administrator Manual* for further details.

Equivalencing of Data

The 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 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 4-4 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 even byte alignment on HP-UX. For performance reasons, 4 or 8 byte data types are normally further restricted to four or 8 byte alignment. If it is necessary to use the minimum even byte alignment because of EQUIVALENCE statement structure, both Series 300 and Series 800 have a `+A` compile line option. In addition, the Series 800 has an `HP1000 ALIGNMENT ON` inline compiler option that will cause data storage to use the minimum even byte alignment for multibyte data types. There are performance penalties incurred when these options are in effect. Memory references to minimally aligned data can slow 0-20% on a Series 300 and 0-200% on a Series 800 when these options are used. Since FORTRAN allows for separate compilation of different program units, it is advisable to compile only the minimum number of program units with these options turned on. Program units that share COMMON areas should be compiled consistently with respect to the alignment options.

For example, the following program will not compile on either the Series 300 or the Series 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` is specified on a Series 300 or 800 or `$HP1000 ALIGNMENT ON` is specified on a Series 800, the program will compile and produce the same results.

VMS-Style records are supported on Series 300 and 800 FORTRAN. This makes interlanguage communication easier. If VMS alignment of structure members is required, specify the `NOSTANDARD ALIGNMENT` option. This option has no effect on the byte ordering of data within structure fields.

See Table 4-4 for specific alignment requirements for Series 300 and 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. In 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 support 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 bug sometime during the life of the program. 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 in the range of 0 to 5% degradation. Since data is staticly stored using this option, your program will have a larger disk image as well. Second, 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 an 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 all system routine names and library names contain at least one lower case letter, name conflicts are thereby avoided.

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. The one exception on HP-UX is `/dev/null`, which is the "NULL" device or bit bucket.

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 and Miscellaneous Tools* tutorial (the Bourne Shell section) from *HP-UX Concepts and Tutorials* 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 in (`stdin`), standard out (`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, it 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, the Series 800 compiler allows 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 Series 800 reference manual for the connected file name. 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 as you desire. If so, the preconnections are closed in accordance with ANSI. Stdin, stdout, and stderr are reconnected when the newly assigned file is closed.

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
```

The following table shows the rough correspondence with VMS predefined logical file names:

Table 2-3. 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) Current directory is used unless an absolute path name is included.

The Pascal Language

No information is currently available on VMS Pascal.

3

Porting from BSD4.3 to HP-UX

No information is currently available on BSD4.3.

4

Porting Across HP-UX

While HP-UX is highly standardized, it is not entirely the same on all machines or implementations. Not all system or language features are present on every implementation, nor is it possible to isolate all hardware architecture characteristics from a user program. This chapter can help you port programs from one HP-UX implementation to another. It presents information about the internal organization of data structures and language features used by C, FORTRAN, and Pascal on all HP-UX implementations.

FORTRAN and Pascal programs must deal with porting across languages since HP-UX is generally a C implementation. HP-UX provides library support, but it is not always written in the same language as your own program. Consequently, this chapter addresses interlanguage communication.

Interlanguage communication is generally based on an *external* routine concept. That is, communication between routines is usually through parameter lists and function results. It is not possible to have routines written in another language access local data except through parameter lists, and in the case of Pascal, the scoping of global routines written in another language is not supported. Given this constraint, considerable effort has been made to make routine calling protocols compatible across language boundaries.

Input/output operations are best performed in the language of the main program because each language has startup code that is specific to input/output initialization. Methods exist to do input/output from external routines, but they are not generic. 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, so it is recommended that input/output be done in the base language.

Unless otherwise noted, the material discussed in this chapter pertains to all HP-UX systems except the Integral Personal Computer.

General System Dependencies

Certain architectural dependencies affect more than one language. This section describes such areas.

Identifying the system

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. See also section "Run Time Hardware Identification" in Chapter 2 for distinguishing floating point hardware on Series 300.

Parameter Lists

On the Series 300, parameter lists grow towards higher addresses. To use a pointer to step through a parameter list, increment the pointer as shown in the following C example:

```
parprint (a,b,c)
        int a,b,c;
{
        int i, *ptr;

        ptr = &a;  /* SET POINTER TO ADD. OF FIRST PARAM */
        for (i = 1; i <= 3; i++)    /* PRINT EACH PARAM */
                {
                printf("%d\n",*ptr);
                ++ptr;
                }

}  /* END parprint */
```

Calling this function would print its three parameters in order.

On the Series 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.

For portability in C, it is highly recommended that variable argument lists be handled using `varargs`. See `vprintf(3S)` and `varargs(5)` in the *HP-UX Reference* for details and examples of `varargs` use.

Parameter passing mechanisms are highly implementation dependent across the HP-UX systems. In most cases these mechanisms are invisible to you unless you are calling routines in another language. See the *HP-UX Assembler Reference Manual and ADB Tutorial* for Series 300 for a further description for this case on Series 300.

Memory Organization

On HP-UX computers, the most significant byte of a datum has the lowest address. This is the address used to access the datum as shown in Figure 4-1.

The conventions of bit numbering differs on HP-UX implementations. On Series 300, the most significant bit of a long word is bit 31 whereas on the Series 800, the most significant bit is bit 0.

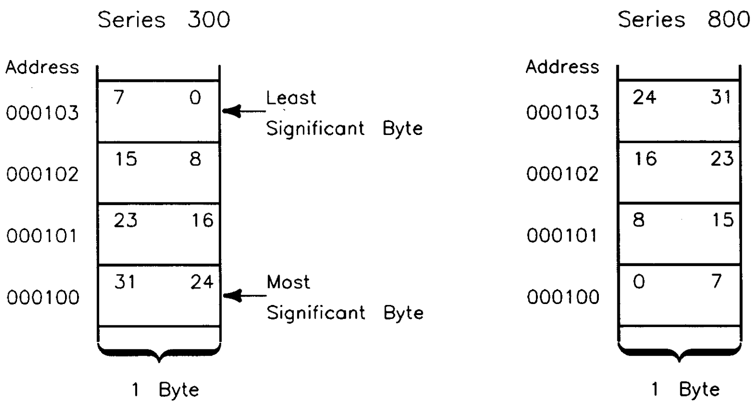

Figure 4-1. Memory Organization

Code/Data Size Limitations

HP-UX systems are not limited in code or data size except by system configuration parameters and file system capacity. These are adjustable by your system administrator. See the *HP-UX System Administrator Manual* for each system for details.

Linker differences

HP-UX compilers and those on other implementations of UNIX generate object files that are eventually linked together to form an executable program. Included in these object files is space for initialized data. On the Series 300, if a global data item (such as a COMMON region in FORTRAN) is declared in two or more files, the size allocated for that data item will be 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 800 linker, on the other hand, will adjust the allocated space to fit the largest declaration.

Optimization

All HP-UX C and FORTRAN compilers will perform those optimizations that are most effective on the particular architecture of the system. However, you cannot assume that the same optimization techniques will be employed on all HP-UX systems.

Beginning with release 6.5 the Series 300 C and FORTRAN 68020/68030 compilers have added an optional global optimization pass to the compilation path. If `-O` or `+O2` is specified on the `f77` command line the global optimization pass will be enabled. `-O` on previous Series 300 releases enabled peephole optimization only; this is equivalent to specifying `+O1` on the 6.5 release. `+O3` enables the global optimizer and the procedure integrator.

Series 800 C, FORTRAN and Pascal compilers have always had global optimization. `-O` on Series 800 is roughly equivalent to `-O` on Series 300 although specific optimization techniques may differ between the two machines. Procedure integration is not performed.

The C Language

To write portable programs in C, give attention to data sizes, parameter passing conventions, and the exact specification of some operations. To avoid subtle errors, be sure the system you move your programs to behaves in expected ways. The next several sections describe areas where the HP-UX implementation of C *may* deviate from other C compilers.

Data Type Sizes and Alignments

This table shows the sizes and alignments of the C data types on the different architectures: (On the 300, this information applies to revision 5.15 and later.)

Table 4–1. C Data Types

Type	Size	Alignment (300)	Alignment (800)
char	8 bits	byte	byte
short	16 bits	2 byte	2 byte
int	32 bits	4 byte (2 byte in a struct, array, or union)	4 byte
long	32 bits	4 byte (2 byte in a struct, array, or union)	4 byte
float	32 bits	4 byte (2 byte in a struct, array, or union)	4 byte
double	64 bits	4 byte (2 byte in a struct, array, or union)	8 byte
pointer	32 bits	4 byte (2 byte in a struct, array, or union)	4 byte
struct/union		4 byte (2 byte in a struct, array, or union)	1,2,4 or 8 byte, depending on types of members

The `typedef` facility is the easiest 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.

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 `MYINT`, and insert the appropriate `typedef`. This would be:

```
typedef long MYINT
```

in code for system A, and would be:

```
typedef int MYINT
```

in code for system B. `#include` files are useful for isolating the system dependent code like these type definitions. 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`.

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 Data Type

The `register` storage class is supported on Series 300 and 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 Reference Manual and ADB Tutorial* for Series 300 for a more complete description of the use of the `register` storage class on Series 300.

Identifiers

Identifiers can be as long as you want, but they have 255 *significant* characters. For universally portable code to non HP-UX systems, use considerably less than this. Eight significant characters for internal identifiers and six for external identifiers (identifiers that are defined in another source file) are safe. Typical C programming practice is to name variables with all lower-case letters, and `#define` constants with all upper case.

Predefined Symbols

The following words are predefined on HP-UX: `hp9000s200` and `hp9000s300` on Series 300 and `hp9000s800` on Series 800. `hpux` and `unix` are predefined on all HP-UX systems.

Note When C becomes standardized, predefined symbols may be disallowed. In that event these symbols will not be predefined for standard conforming programs.

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.

Sizeof

The sizeof operator yields an unsigned result. Therefore, expressions involving this operator are inherently unsigned. Do not expect any expression involving the sizeof operator to have a negative value; in particular, logical comparisons of such an expression against zero may not produce the object code you expect (See the following example).

```
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

Bit fields are assigned left to right and are unsigned regardless of the declared type on Series 300 but they can be signed on Series 800. They are aligned so they do not violate the alignment restriction 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 bitfields 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)` would return 6 (bytes).

Bitfields on HP-UX systems cannot exceed the size of the declared type in length. The largest possible bitfield is 32 bits. All scalar types are permissible to declare bitfields, including `enum`.

`Enum` bitfields are accepted on all HP-UX systems. On Series 300 they are implemented internally as unsigned integers. On Series 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.

Division by Zero

Division by zero gives the run time error message `Floating exception (core dumped)`.

Integer Overflow

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 runtime if a floating point number is converted to an integral type and the value is outside the range of that integral type.

Structure Assignment and Functions

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 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;
```

Null Pointers

Accessing the object of a null pointer is technically illegal. However, some versions of C permit references to null pointers. If you try to read using a null pointer on HP-UX, a value of zero is returned. The Series 800 compiler recognizes the `-z` option which causes a run time error to be produced instead. Since some programs written on other implementations of UNIX rely on being able to reference 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')
```

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

Expression Evaluation

The order of evaluation for some expressions will differ between HP-UX computers. 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, avoid relying on the order of evaluation when using functions with side effects and function calls as actual parameters. Use temporary variables if your program relies upon a certain order of evaluation.

Variable Initialization

On some C implementations, `auto` 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 variables; it is not good programming practice in general and it makes for nonportable code.

Conversions

All HP-UX C implementations are `unsign 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` (e.g. `unsigned char` terms are first converted to `char` and then to `int` before they are used in an expression).

The following program will print:

```
Unsigned preserving
Unsigned comparisons performed
```

on HP-UX systems (and most other C implementations):

```
  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");
  }
```

$TMPDIR

HP-UX 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.

Compiler Command Options

There are some minor differences between HP-UX C compiler options. If you are using make, you may have to change the compile lines in your makefiles when porting your code. Here is a list of the variant options. See the *HP-UX Reference* for more details. Series 800 supports all these options.

Table 4-2. Differences in C Compiler Command Line Options

Option	Effect	Difference
-G	Enable G profiling	Supported on Series 300 only.
-W	Pass options to subprocesses.	System dependent options. See cc(1) in the *HP-UX Reference* for details.
+<option>	Shorthand for -W	System dependent options. See cc(1) in the *HP-UX Reference* for details.
-Z	Allow dereferencing of null ·detairs.	Has no effect on Series 300 pointers.
-z	Allow run time detection of null pointers.	Not supported on Series 300.

Calls to 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. The ramifications of this depend on the language of the called function (See Table 4-3).

Table 4-3. C Interfacing Compatibility

C	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 800)	integer*2
unsigned short (int)	none (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
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)	record (Series 300 only)
union	record case of ...	equivalence

Calls to 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 all parameters are passed by reference 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.

The FORTRAN function and BLOCK DATA subprogram contained in file `xx.f` are compiled using `f77 -o xx.f`:

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

      block data one
      double precision array
      common /a/array(1000,10)
C Note how easy 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 C main program contained in file `x.c` is then compiled using:

```
cc x.c xx.o:


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

Calling FORTRAN subprograms from other languages presents special problems if the subprograms do any I/O. In particular, file handling for FORTRAN requires special startup code and exit code generally provided by the `frt0.o` link file. A program that mixes I/O from FORTRAN subprograms and functions from other languages is not recommended if the main program is not also in FORTRAN.

Calls to Pascal

Pascal gives you the choice of passing parameters *by value* or *by reference* (`var` parameters). C passes all parameters 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.

The one exception to the *pass by value* parameter passing mode in C is when an array is used as a parameter. In this case, only the address of the first element of the array is actually passed. To pass the the array by value, it is necessary to enclose the array within a `struct` type. 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 (as in FORTRAN long logicals). Also, C performs all scalar 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 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 or FORTRAN on the Series 800, the user must first call the Pascal procedure `U_INIT_TRAPS`. See the *HP Pascal Programmer's Guide* for details about the try-recover mechanism.

To call Pascal procedures from C or FORTRAN on the Series 300, 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. The trick to making this work is to call **asm_initproc** 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. The following page has an example.

The Series 300 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 source below is for 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.
```

The command line for producing the Pascal relocatable object is

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

The command line on Series 300 for compiling the C main program and linking the Pascal module is then

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

or on Series 800 is

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

Which produces the following output:

```
2147483647
-2147483648
```

The FORTRAN Language

All HP-UX FORTRAN compilers implement the full ANSI FORTRAN 77 language and MIL-STD-1753 extensions. In addition, many common extensions found in other NON-HP-UX implementations have been added, particularly those from FORTRAN 7x on HP1000 systems and VAX VMS FORTRAN. See Chapter 2 for further details on VAX VMS feature extensions.

Data Type Sizes and Alignment

This table shows the sizes and alignments of the FORTRAN data types on the different architectures:

Table 4-4. FORTRAN Data Types

Type	Size	Alignment (300)	Alignment (800)
character	8 bits	2 byte	1 byte
Hollerith[1] (character)	8 bits	2 byte	1 byte
byte,logical*1[1/2/3]	8 bits	2 byte	1 byte
logical*2[1/2]	16 bits	2 byte	2 byte
integer*2[1/2]	16 bits	2 byte	2 byte
logical (*4)[2]	32 bits	4 byte (2 byte with +A option)	4 byte (2 byte with $HP1000 ALIGNMENT ON)
integer (*4)[2]	32 bits	4 byte (2 byte with +A option)	4 byte (2 byte with $HP1000 ALIGNMENT ON)
real (*4)[2]	32 bits	4 byte (2 byte with +A option)	4 byte (2 byte with $HP1000 ALIGNMENT ON)
real (*16)[1/2]	128 bits	Not supported	8 byte
double precision,real*8[2/3]	64 bits	4 byte (2 byte with +A option)	8 byte (2 byte with $HP1000 ALIGNMENT ON)
complex (*8)[2]	64 bits	4 byte (2 byte with +A option)	4 byte (2 byte with $HP1000 ALIGNMENT ON)
double complex,complex*16[1/2/3]	128 bits	4 byte (2 byte with +A option)	8 byte (2 byte with $HP1000 ALIGNMENT ON)
record		4 byte (2 byte in array or another record; alignment alterable using NOSTANDARD ALIGNMENT)	Aligned on most restrictive field

1 This type is an extension to ANSI FORTRAN77.
2 ANSI does not support a length descriptor "*n ".
3 Synonymous types.

Alignment requirements for the larger data types can be reduced via the `+A` compiler option on Series 300 and the `$HP1000 ALIGNMENT ON` inline option on the Series 800. On the Series 800 `ALIGNMENT` has the `HP9000_300` and `HP9000_500` options" available. See Chapter 2 section "Data Representations in Memory" for further details.

Long Identifiers

All HP-UX implementations allow identifiers to be at least 255 characters long with the first 255 being significant.

Error Conditions

The various HP-UX FORTRAN compilers are based on different technologies. So, it is not possible to detect compile time error conditions in the same ways on each. Since some errors are detected at compile time on a Series 300, that might be detected at link time on a Series 800, compile time error messages are not compatible between the systems. Series 300 gives plain text error messages. Series 800 gives error numbers with optional text messages in a different format.

Run time errors are much more compatible because both systems use a common set of run time libraries. In most cases run time errors will be reported with the same message and number on all HP-UX systems. Some exceptions may be seen when arithmetic overflow/underflow conditions occur. On Series 300, the various floating point options may cause slightly different arithmetic error condition response at corner cases.

HP-UX compilers are more relaxed about statement sequencing than ANSI. In many cases duplicate declarations are allowed, although the result may be undefined if they are conflicting. Series 800 will issue a warning message, `DUPLICATE DECLARATION OR DEFINITION, USING FIRST TYPE (767)`.

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

```
$OPTION ANSI ON
```

at the beginning of your source will make the compiler include in the listing warnings for uses of features that are 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 on Series 300, you will get a non-fatal warning for each lower case letter, possibly resulting in a large `stderr` file. The error will be written only once per function on Series 800.

String Constants

String constants are limited to 9000 characters in length on Series 300 whereas they are essentially unlimited on Series 800. If a longer constant is required on the Series 300, it can be constructed by use of the // concatenation operator. Such concatenated strings have no length restrictions.

Array Dimension Limits

While ANSI requires that FORTRAN implementations support at least 7 dimensions, Series 300 permits up to 20. Series 800 makes no restrictions on the number of array dimensions.

Data File Compatibility

Since all HP-UX FORTRAN implementations use the same 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. Even they can be accessed compatibly if the appropriate alignment options are set when they are written. 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 with this exception; on the Series 800 a compiler directive or `HP9000_300 LOGICALS` has to be used to make the internal representations of logical values the same as those on Series 300.

```
      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.

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

Parameter Passing

All HP-UX FORTRAN compilers have implemented the `$alias` directive which allows you to change addressing modes when calling routines in other languages. The statement is source compatible across the HP-UX line with the exception that `%descr`, which forces `pass-by-descriptor` addressing, has no meaning on Series 300. `%descr` will be flagged as a fatal error on that system.

Series 800 FORTRAN normally uses pass-by-descriptor conventions when passing character strings through parameters. The `HP9000_300 CHARS` or `HP9000 CHARS` directives can be used to pass them in the same manner as is used on the Series 300. That is, the string length is passed as a hidden parameter at the end of the parameter list, and the string is passed by reference.

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. User programs which use the same name for a common region and a subprogram will not run correctly on the Series 300 unless the `RENAME_COMMON` directive is specified. This directive is not necessary on the Series 800 because the compiler supports the extension by default.

Note When using the `RENAME_COMMON` directive, the Series 300 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

The following vector instruction set subroutines are supported on HP-UX:

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

Compiler Options

The HP-UX FORTRAN compilers support different command line options. Table 4-5 on the following page has a list of the options that vary between the systems. Options that are the same on both systems are not listed here. See the *HP-UX Reference* for more details.

Table 4-5. Differences in FORTRAN Compiler Command Lines

Option	Effect	Difference
-A	Specify ANSI warning level	S300 only
+A	Force 2-byte data alignment	S300 supports additional alignment modes.
+B	Special handling of backslash ("\")	S300 only
+bfpa	Floating point option	S300 only.
+E3	Enable VMS character passing	S800 only
+E4	Enable VMS I/O on format specifiers `A` and `R`	S800 only
+ffpa	Floating point option	S300 only
-G	Berkeley style profiling	S300 only
-K	force static allocation	S300 has side effects
+M	Floating point option	S300 only
+N	Adjust table sizes	S300 only
-O	Specify optimization level	only S800 allows a qualifier
+O	Specify optimization level	S300 only
+P	Invoke procedure integrator	S300 only
-R	Specify real constant default sizes	S300 only
+T	Procedure traceback	S800 only
-u	Implicit typing off	Can be overridden in a program unit on S300
+U	Case is significant	S300 only
-w66	Suppress FORTRAN 66 warnings	S300 only
-y	Static analysis option	S800 only

Compiler Directives

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

Directives Only on Series 300

HP9000_800 ALIGNMENT	INLINE	NOSTANDARD ALIGNMENT	RENAME_COMMON

Directives Only on Series 800

CHECK_OVERFLOW	EXTERNAL_ALIAS	INIT	SYSINTR
CODE	FTN3000_66	LIST_CODE	SYSTEM INTRINSIC
CODE_OFFSETS	GPROF	LOCALITY	VERSION
CONTINUATIONS	HP1000	NOSTANDARD IO	
COPYRIGHT	HP9000_500	PAGEWIDTH	
DEBUG	HP9000 CHARS	SET	
ELSE	HP9000 LOGICALS	STANDARD_LEVEL	
ENDIF	IF	SYMDEBUG	

The ALIAS Directive

Series 300 does not allow the %desc passing mode with the ALIAS directive.

The OPTIMIZE Directive

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

The SAVE_LOCALS Directive

The default settings for the SAVE_LOCALS directive vary by implementation to preserve backward compatibility.

Recursion

One major feature of HP's versions of FORTRAN is that they support recursion. This means that variable storage for subroutines and functions is dynamic. Hence, variables in subprograms do not retain their values between invocations.

For a more detailed discussion on the effects of recursion and some debugging hints, see Chapter 2, "The Effects of Recursion on Local Variable Storage".

$TMPDIR

Series 300 and 800 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 the requirements for these files, which are usually generated on `/tmp` or `/usr/tmp`, may exceed the space available. By assigning another directory name to the `TMPDIR` environment variable you can redirect these temporary files. See the `f77` manual page for details.

Calls to Other Languages

Most of the comments made earlier 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 4-6).

Table 4–6. FORTRAN Interfacing Compatibility

FORTRAN	Pascal	C
character	char[1]	unsigned char[1]
Hollerith (synonymous with character; extension to ANSI FORTRAN77)		
byte, logical*1[2](extension to ANSI FORTRAN77; synonymous types)	-128..127 or boolean	char[1]
logical*2[2] (extension to ANSI FORTRAN77)	-32768..32767	short
integer*2 [2] (extension to ANSI FORTRAN77)	-32768..32767	short
logical (*4)[2]	integer	long or int
integer (*4)[2]	integer	long or int
real (*4)[2]	real	float
double precision, real*8[2] (synonymous types)	longreal	double
complex (*8)[2]	record	struct
double complex, complex*16[2] (extension to ANSI FORTRAN77; synonymous types)	record	struct
record	record	struct

1 FORTRAN can pass its characters to Pascal and C, but when calling in the other direction the character type may reside on an odd boundary and cause a memory fault.

2 ANSI does not support a length descriptor *n.

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 *Pascal Language 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 VMS-style `records` differs from C `structs`, default packing and alignment rules are similar between the two languages.

Another caution in interfacing FORTRAN with C or Pascal stems from the fact that 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. It is best to explicitly declare the length at all times.

Calls to 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 (See the example):

```
      PROGRAM TESTALARM

$ ALIAS IALARM = 'alarm'(%val)

C set a 10 minute alarm
          I <C=:C4<A47=7  ϒ¿ ¿æ+- '(C
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
```

Note these items:

Logicals	C uses integers for logical types. A FORTRAN 2-byte LOGICAL is equivalent to a C `short`, and a 4-byte LOGICAL by a `long` or `int`. In both C and FORTRAN, zero is false and any non-zero value is true.
Files	File units and pointers can be passed 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. To 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.

```
#define MSLEN 300

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

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

```
$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
```

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* 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)

                       Øfor aER 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
```

The Pascal Language

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 and 800 which are covered in this section. Another source of information, particularly for Series 800, is the *HP Pascal/HP-UX Migration Guide.*

The extensions of HP Pascal are a blessing and a curse. If you plan only to run your programs on HP computers (better yet, only HP 9000 computers), then it won't take much work to move them, and the extra features will make your programming much easier. *However,* if you should decide to port those programs to another manufacturer's computer, the effort to do so will be proportional to the use of non-standard Pascal extensions. Even if the system you are moving the programs to has extensions, it is doubtful that they have the same form as HP Pascal. Before deciding to use a non-ANSI feature, ask yourself some questions:

- Am I *ever* going to port this program to a non-HP system?
- How much hardship does avoiding the extension cause?
- Will another system have a similar feature?

If your answers are something like "probably not," "a lot," and "I sure hope so," then go ahead and use the extension.

How can you know whether any of the language features you are using are likely to be supported on another system? HP Pascal has an option that causes the compiler to emit errors for uses of features not included in ANSI Standard Pascal. On all HP-UX systems, include the line

```
$ANSI ON$
```

at the beginning of your source file. You will have to use the `-L` option with `pc` and look at a listing of your program (on the screen or hardcopy) to see where the warnings occurred.

Data Type Sizes and Alignments

Table 4-7 shows the sizes and alignments of the Pascal data types on HP-UX architectures. For more specific information, see the appropriate Language Reference or the *HP Pascal Programmer's Guide.*

Table 4-7. Pascal Data Types

Type	Size	Alignment (300)	Alignment (800)
`char`	8 bits	1 byte	1 byte
`boolean`	8 bits	1 byte	1 byte
`shortint`	16 bits	Not supported	2 byte
subrange of `integer`	16 bits[1/2]	2 byte	2 byte or 4 byte, based on declared size
`integer`	32 bits	4 byte (2 byte with `+A` option)	4 byte
`longint`	64 bits	Not supported	4 byte
`enumeration`	16 bits[2]	2 byte	2 byte or 4 byte, based on declared size
subrange of `enumeration`	16 bits[2]	2 byte	2 byte or 4 byte, based on declared size
`real`	32 bits	4 byte (2 byte with `+A` option)	4 byte
`longreal`	64 bits	4 byte (2 byte with `+A` option)	8 byte
pointer	32 bits	4 byte (2 byte with `+A` option)	4 byte
`set`	Varies	Varies	Varies

1 On Series 300, allocation is 16 or 32 bits based on the declared size.

2 On Series 800, allocation can be 8, 16, or 32 bits based on the declared size.

Command Line Options

Table 4-8 shows some differences in the Pascal compiler (`pc`) command between the HP-UX implementations. See the *HP-UX Reference,* `pc(1)` for details.

Table 4-8. Differences in Pascal Compiler Command Lines

Option	Effect	Difference
`+A`	use 2-byte alignment rules	Series 300 only
`-L`	produce a program listing	Series 300 goes to a specified file
`+M`	library calls for floating point	Series 300 only
`+O`	optimization	Series 800 only
`-O`	optimize	Series 800 only

Inline Compiler Option Differences

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.

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

ALIAS	Series 300 does not automatically add an underscore (_) prefix. Series 800 does.
ALIGNMENT	Series 800 only. Changes storage alignment for types other than strings and file types.
ALLOW_PACKED	Series 300 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 requires that it be at the top of the file.
ASSERT_HALT	Series 800 only. Causes the program to halt if the `assert` function fails.
ASSUME	Series 800 only. Sets or lists optimizer assumptions.
BUILDINT	Series 800 only. Causes the compiler to build an intrinsic file rather than an object code file.
CHECK_ACTUAL_PARM	Series 800 only. Sets level of type checking of actual parameters for separately compiled functions or procedures.
CHECK_FORMAL_PARM	Series 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 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 disallows this directive within a procedure body.
COPYRIGHT	Series 800 only. Causes a copyright string to be placed into object code.

COPYRIGHT_DATE	Series 800 only. Sets the copyright year and causes it and the copyright string to be placed into object code.
DEBUG	Series 300 only. Causes line number debugging information to be included in the relocatable file.
EXTERNAL	Series 800 only. Used in conjunction with the GLOBAL option, enables you to compile one program as two or more compilation units.
FLOAT_HDW	Series 300 only. Controls generation of code for floating point hardware.
GLOBAL	Series 800 only. Used in conjunction with the EXTERNAL option, enables you to compile one program as two or more compilation units.
HEAP_COMPACT	Series 800 only. When this and HEAP_DISPOSE is on, free space in the heap is concatenated.
HEAP_DISPOSE	Series 800 only. Disposed space in the heap is freed for new uses by `new`.
IF, ELSE, ENDIF	Available on all HP-UX implementations. Controls conditional compilation. While the basic semantics are the same, each implementation has minor differences in semantics. Refer to the appropriate Language Reference for details. ELSE is permitted in conjunction with IF on Series 800. ENDIF is required as a terminator.
INLINE	Series 800 only. Causes a procedure call to be replaced by inline code.
KEEPASMB	Series 800 only. Causes the compiler to preserve an assembly file for the source file.
LINENUM	Series 300 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, and 59 for Series 800.

LIST_CODE	Series 800 only. When LIST is also on, a mnemonic listing of object code is produced.
LISTINTR	Series 800 only. List an intrinsic file to a specified file.
LITERAL_ALIAS	Series 800 only. Changes the semantics for the ALIAS option.
LOCALITY	Series 800 only. Causes a locality name to be written to the object file for performance enhancement.
LONGSTRINGS	Series 300 only. Extends the maximum length of strings from 255 to virtually unlimited.
MLIBRARY	Series 800 only. Specifies alternate file into which the modeule export text is to be written.
NOTES	Series 800 only. Causes helpful compiler notes to be printed on the program listing.
OPTIMIZE	Series 800 only. Sets level of optimization.
OS	Series 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 800 only. Changes external names to a form consistent with Series 300 conventions.
SAVE_CONST	Series 300 only. Controls scope of structured constants.
SEARCH	Available on all HP-UX implementations. Series 800 has two ways to create the list of files (one of which is the same as Series 800, the other uses MLIBRARY).
SEARCH_SIZE	Series 300 only. Changes number of external files that can be searched. The default is 9.

SKIP_TEXT	Series 800 only. Causes the compiler to ignore source code.
SPLINTR	Series 800 only. Specifies what intrinsic file is to be read.
STATEMENT_NUMBER	Series 800 only. When enabled, the compiler generates a special instruction to identify a code sequence with its corresponding Pascal statement.
STRINGTEMPLIMIT	Series 300 only. Specifies the maximum size of a temporary string used within a string expression.
SUBPROGRAM	Series 800 only. Separate compilation facility. Use modules instead.
SYSINTR	Series 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 forbids its use within a procedure body whereas Series 800 permits it anywhere.
TITLE	Series 800 only. Specifies the title to appear on the program listing.
UNDERSCORE	Series 300 only. Causes ALIAS parameters to have an underscore added as a prefix.
UPPERCASE	Series 800 only. All external names are shifted to upper case ASCII.
VERSION	Series 800 only. Specifies a version stamp to be placed in the object file.

Differences in Features

Due to the varying origins of the HP-UX Pascal compilers, there are some differences between them. Here is a list of the features that differ between Series 300 and 800 HP-UX Pascal.

Control Constructs

- Try-Recover is supported on all HP-UX implementations. Escape codes for errors differ between the implementations.
- Mark/Release is supported on all HP-UX implementations. There are minor differences in behavior but code is essentially portable.

I/O

- The `maxpos()` function always returns `maxint` on the Series 300.
- Series 300 and 800 differ in the way they allow association with an HP-UX file descriptor in the `reset()` procedure. The association is now similar in the `associate()` procedure.
- Series 800 uses the `options string` parameter on `reset()`, `rewrite()`, `open()`, and `append()` procedures. Series 300 ignores this parameter.
- By default, `stdout` is buffered on the Series 800. It can be changed to unbuffered via an option.

Program Structure

- Modules are supported on all HP-UX implementations but some syntactic and semantic differences exist. For example, Series 800 requires that CONST, TYPE, and VAR declarations precede routine declarations within the EXPORT section whereas Series 300 permits them to be intermixed.
- Series 300 permits separate compilation only within modules. Series 800 provides other mechanisms as well.

Types

- Assignments to procedure variables have a different syntax on the two systems.
- On the Series 300, the maximum string size is 255 characters by default but by specifying `$LONGSTRINGS$` it can be virtually unlimited. Series 800 strings are essentially unlimited.
- On the Series 300, elements of packed arrays can be passed as `anyvar` parameters only if the ALLOW_PACKED compiler option has been used.
- All HP-UX implementations support structured constants but different restrictions may apply. Series 300 restricts their use to within the CONST section and it does not do full type checking on variant record structured constants.
- A small difference in precision exists between the implementations of `longreal`.
- Only Series 800 implements `globalanyptr` and `localanyptr`. All HP-UX implementations implementations have `anyptr`, although minor differences exist.
- `Anyvar` is supported on all HP-UX implementations. Series 300 does not perform any checks to see if `anyvar` values are legitimate.

Miscellaneous

- Series 800 supports `readonly` parameters. Series 300 does not.
- Series 300 Pascal allows using a file variable as a parameter to the `sizeof` function; Series 800 does not.
- Only Series 800 supports `crunched` arrays and records.
- Series 800 does not fully support `packed array[0..`<anything>`] of char`.
- Slight semantic differences exist between Series 300 and 800 program parameters.

Calls to 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 4-9.

Table 4-9. Pascal Interfacing Compatibility

Pascal	C	FORTRAN
`boolean`	`unsigned char`	`logical*1` (`logical*2` and `logical*4` won't work)
`char`	`char`	`character*1` (the Pascal `char` must not lie on an odd byte)
`integer`	`long; int`	`integer` (*4)
-32768..32767	`short`	`integer` (*2)(extension to ANSI standard FORTRAN77)
`real`	`float`	`real` (*4)
`longreal`	`double`	`double precision`
enumerated type	`enum`	use `integer*2` (extension to ANSI standard FORTRAN77)
subrange (32-bit)	use `long; int`	use `integer*4`
subrange (16-bit)	use `short`	use `integer*2` (extension to ANSI standard FORTRAN77)
`set`	none	none
`record`	`struct` (the fields must align)	`record`
^<`type`> (pointer)	`type *; &var`	none
<`var`>^ (dereferencing)	`*var`	none

Take care when using packed records in Pascal, in that the compiler packs the data into the smallest required space. Thus, fields may not align on byte boundaries. This makes it extremely 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.

On Series 300, to call an external (FORTRAN, C, or assembler) procedure, the user must declare a Pascal interface to it, and then define it as `EXTERNAL`. Pascal will then add a _ 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 doesn't 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 800 except that the underscore is neither added nor required for external names.

Calls to C

HP-UX system calls and subroutines are defined as C functions, so you may need to call a C function from a Pascal program. Fortunately, Pascal and HP-UX are flexible enough to make this a simple operation. This section contains a list of concerns 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 can emulate pass by reference by declaring a formal parameter to be 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. Don't assume that it works just because the function works when called by a program in the same language.
- If you want to refer to an external function by a name other than the one it is defined under, use the `alias` directive.

This example shows how to call a user defined C function from a Pascal program. First is 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.
```

Next 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 procedure for compiling and linking these two source files is:

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

Then, executing the file named `a.out` would 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
```

5

System Calls and Subroutines

This chapter explains differences between system calls and subroutines among the HP-UX implementations. If you have hardware dependencies or need more information about a routine, see the *HP-UX Reference* for your system.

On Series 800 only, many of the system calls have separate entry points suitable for call from FORTRAN. See the *HP-UX Reference* for details.

System Calls

`acct`	Many Series 300 system dependencies exist.
`exec`	Unsharable executable files are not supported on Series 800.
`gettimeofday`	Series 300 has a granularity of 4 microseconds.
`ioctl`	Asynchronous I/O is supported only on Series 800.
`mknod`	On Series 300 the additional value *0110000* is available under file type and specifies network special files.
`ptrace`	The functionality of `ptrace` is dependent on the hardware and may not be portable.
`reboot`	Many differences exist.
`select`	Many system differences exist.
`shmctl`	Series 300 has differences in the handling of EACCESS.
`shmop`	Implementation differences exist between Series 300 and 800 involving `shmaddr` and some variables and constants.
`signal`	Implementation differences exist between Series 300 and 800.
`sigspace`	On Series 300, guaranteed space is allocated with `malloc` and may interfere with other heap management efforts. Also, include kernel overhead in any requested amount of space.
`sigvector`	The SV_BSDSIG flag is not supported on Series 300. The SC_RESETHAND flag is not supported on Series 800.
`uname`	The `U` version field is not supported on Series 300.
`ustat`	On Series 300, `f_tfree` and `f_flksize` are reported in fragment size units.
`vfork`	Programs which rely upon the differences between `fork` and `vfork` may not be portable across HP-UX systems.
`vfsmount`	`Vfsmount` of a local file system on a diskless Series 300 node is not supported.

Subroutines

`abs`	On HP-UX, calling `abs` with the most negative number returns that number.
`blclose, blread, blget, blset`	Series 800 only.
`clock`	Clock resolution is 20 milliseconds on Series 300, 10 milliseconds on Series 800.
`dial, undial`	`Dial` is not supported on client nodes of an HP Cluster. Series 300 has system dependencies.
`ftok`	System dependencies on Series 300 diskless only.
`gpio_get_status, gpio_set_ctl`	System dependencies exist.
`hpib_abort, hpib_bus_status, hpib_eoi_ctl, hpib_rqst_srvce`	System dependencies exist.
`hpimage`	Series 800 only
`io_burst, io_lock, io_unlock`	Series 300 only.
`io_get_term_reason, io_on_interrupt, io_reset, io_speed_ctl, io_timeout_ctl`	System dependencies exist.
`syslog, openlog, closelog, setlogmask`	Series 800 only.
`_toupper, _tolower`	Series 300 does not define the results for non-ASCII arguments.
`trig`	On Series 300 the approximate limit for values returned by the `trig` functions is 1.49^8.

6

Pascal Workstation to HP-UX

This chapter helps you port programs from the Series 200/300 Pascal Workstation to Series 300 HP-UX. It focuses on conversions of Pascal programs, but has some comments on assembly language translation. The information applies to a Series 300 HP-UX system. Thus, some of the comments may not apply to Series 800 porting. The topics deal with the commonly encountered porting problems.

Since the Series 300 HP-UX Pascal compiler was developed from the Series 200/300 HP Pascal workstation, the two implementations are very similar. There are still some differences for you to note in 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 non-trivial porting job.

The chapter does not cover the differences between Series 200 and 300 workstations. The few differences that exist are documented in the *Pascal 3.1 Workstation System Vol. II: Programming and Configuration Topics,* Chapter 20, "Porting to Series 300".

Compiler Option Differences

The options available on HP-UX Series 300 Pascal are, with three exceptions, 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 workarounds 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, there is one compiler option, PARTIAL_EVAL, which is implemented differently on the two systems. Default on the Pascal Workstation is "OFF", but the default on HP-UX Series 300 Pascal is "ON". This was done to make HP-UX Series 300 Pascal compatible with older HP-UX Pascal implementations. Note that this is different from early releases of Series 300 HP-UX Pascal.

Differences in Features

There are some minor semantic differences between the workstation and HP-UX Pascal implementations. The next several sections describe them.

Module Names

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

Real Variables

Real variables are 32 bits on HP-UX Pascal and 64 bits on the workstation. Longreals are 64 bits on both implementations.

Input

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

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

Lastpos

Not implemented on the Pascal workstation.

Linepos

Not implemented on the Pascal workstation.

Heap Management

The Series 300 HP-UX and Pascal workstation have different mechanisms for specifying the heap manager. See the *HP Pascal Language Reference* for the details of using them.

File Naming

File naming within Pascal programs (e.g. in $INCLUDE statements) on HP-UX must follow HP-UX path naming conventions. File names in programs on the Pascal workstation are of the form:

`VOL:`*FILENAME*

Absolute Addressing

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

```
..m off
   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 `symbol already declared` on `asm_ioresult`.

Accessing absolute addresses (such as the Model 236 graphics display) will result in the system error `segmentation violation`. To gain access to this memory, the user must use the techniques described in the *HP-UX Reference Section 4: Graphics(4)*.

$SEARCH$ File Names

`$SEARCH$` file names (300) must refer to either simple relocatable (`.o`) or archived (`.a`) format object files. Libraries will be maintained by the ARchiver, 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

HP Pascal 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 *&<newline>* . To overcome this system buffering of input into lines, the user must specify :

```
..m off
   rewrite(input,",'unbuffered');
```

File Naming

File naming on HP-UX 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 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')`. Another method of doing this using system calls appears in the section “Example Program.”

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.

On HP-UX, 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 doesn't 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 requires using $sysprog$ and ANYPTRs. However, this is compatible with allocation by system intrinsics and C.

The following program (which covers several pages) shows how to use the system intrinsic IOCTL to modify the terminal characterists. It does unbuffered, non-echoed terminal input. IOCTL turns off echoing, and sets the minimum length line to 1 character, and 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.
  ..m
```

Library Differences

The workstation and HP-UX Pascal use different libraries. This manual will not discuss the differences but refers you to the manuals containing the information on the libraries.

For Pascal workstation library information, see the *Pascal Procedure Library* manual. HP-UX library information is contained in several HP-UX manuals. For Graphics see the applicable graphics manuals The system library is documented in section 3 of *HP-UX Reference.* The I/O library is documented in *Concepts & Tutorials.*

On the Pascal Workstation there are three primary libraries used by almost everyone. The first is the DGL graphics library which provides a high level (Pascal) interface to device-independent graphics. Another library is the I/O library which provides various levels of access to the I/O cards on the Series 300 system. These include HP-IB, GPIO, and a serial interface library. The other library is the `INTERFACE` library. This is a permanently loaded library (via `initlib`), which contains much of the operating system software (disk drivers, keyboard, etc.). Here we will look at some of these same capabilities on HP-UX.

Graphics

DGL Library

Graphics on the Pascal Workstation is performed through a library named DGL. This is a functional copy of the old HP 1000 FORTRAN DGL library. The interface has been changed to provide more meaningful names for the procedures as well as a Pascal interface.

On HP-UX the original HP1000 FORTRAN DGL library was ported and created these differences from the Pascal Workstation:

- The names have reverted to their original names
- Parameters are all passed by reference (`var`)
- Strings are FORTRAN character arrays (with a separate length parameter)
- Integers are 16-bit integers

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

Unfortunately, there are no workarounds for other FORTRAN problems. Since all parameters are passed by `VAR`, all constants must first be assigned to dummy variables. Secondly, 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.

The graphics (DGL) library is not described in the *HP-UX Reference,* but has a manual of its own.

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) will 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 (ARchive) files. These reside in the directories `/lib` and `/usr/lib` as well as in the kernel itself. The capabilities provided exceed that available on the Pascal Workstation in many cases, and in others it falls 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 easy, with the main difficulty coming from replacing the header files that are needed.

Assembly Language Conversion

The conversion of assembly language routines from the Pascal Workstation to HP-UX is fairly easy. An HP-UX command exists on the Series 300 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.

- 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 valid identifier characters on the HP-UX assembler. It is translated to `A` and a warning is issued.
- Lines containing these 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.
- Certain pseudo-ops require manual intervention to translate. Each line containing these pseudo-ops will cause a message to be printed stated that an error will be generated by the HP-UX assembler. These pseudo-ops are: `com`, `lmode`, `org`, `rorg`, `rmode`, `smode`, and `start`.
- When specifying certain addressing modes, the Pascal workstation assembler may allow operands to appear out of order, whereas the HP-UX assembler does not. `Atrans` does not rearrange these into proper order.

Index

HEWLETT PACKARD

HP Part Number
98794-90047

Microfiche No. 98794-99047
Printed in U.S.A. E0989

98794-90647
For Internal Use Only