HP-UX Portability Guide

HP 9000 Series 200/300/500 Computers

HP Part Number 97033-90046

HEWLETT
PACKARD

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

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. A vertical bar in the margin indicates the changes on each 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.

November 1985...Edition 1

May 1986...Update

July 1986...Edition 2. This Edition reflects changes in the Series 300 5.15 release and the Series 500 5.1 release. May 1986 Update incorporated.

July 1987...Edition 3

Table of Contents

Introduction 1

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

The concept of portability has enormous breadth. This manual concentrates on aspects of portability dealing with moving C, FORTRAN, and Pascal source code from one system to another.

On providing a general overview of basic principles of portability, the manual concentrates on the following three areas of interest:

- **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, 500, and 800 systems. Some of this information can be useful to people who port code from another operating system. You get descriptions of interlanguage communication issues. 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.** The list is not complete, but you see 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.

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 co-operating processes (works only in Series 300 systems).

Overall, the manual provides a picture of portability as it exists now. In particular, it refers only to release 5.5 for Series 300, release 5.2 for Series 500, and release 1.1 for Series 800 implementations. Subsequent releases may alter the data contained within this manual. When in doubt about features for FORTRAN or Pascal, refer to the HP-UX 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(TM) System V.2. It passes SVVS validation on Series 300 and 800. Series 500 passes with some exceptions.
- 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.

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

* UNIX is a trademark of AT&T Bell Laboratories, Inc.

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 in HP-UX. This manual will describe the extensions. C programs in general have 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, 500, and 800 FORTRAN compilers fully comply with this standard and all 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 in Series 300, 500 and 800 FORTRAN. All extensions will be described later in the manual.

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, 500, and 800 conform or are converging. Pascal on all these architectures conform 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, there are directives that enable system dependent features that dissolve any hope of portability. In any case, the directives will have to be reviewed when porting because it is unlikely that the systems you are porting between support the same directives. You must balance the current usefulness of the directive against its potential for portability problems.

Parent and Self Directory Entries

On Series 500 HP-UX, directories do not contain entries for . and .. (current and parent directories respectively). Any program that relies on those entries being present will not work.

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

Software engineers who require floating point math performance should account for the following possible hardware configurations of Series 300 systems:

- Series 310 (68010 processor) with no floating point hardware.
- Series 310 (68010 processor) with the HP 98635A floating point math card.
- Series 320 (68020 processor) with 68881 floating point coprocessor.
- Series 330 (68020 processor) with 68881 floating point coprocessor.
- Series 330 (68020 processor) with 68881 floating point coprocessor and the HP 98248A floating point accelerator card.
- Series 350 (68020 processor) with 68881 floating point coprocessor.
- Series 350 (68020 processor) with 68881 floating point coprocessor and the HP 98248A floating point accelerator card.

Series 320, 330, and 350 systems also support the HP 98635A floating point math card, but it is not recommended that you do this because performance with the 68881 coprocessor exceeds the performance and precision available with the card.

The default compilation mode on Series 310 systems for C and FORTRAN is to generate calls to floating point math routines within the standard libraries *libc.a* and *libm.a*. By specifying `+f` as a compiler option, the compiler will instead generate inline references to the HP 98635A floating point math card. This code requires the presence of the card to execute successfully. If `+b` is specified instead, the compiler generates calls to determine the absence or presence of the card at runtime. The floating point math card code will be executed only if the card is present. The advantage of the `+b` option is that one set of bits will run on any Series 300. Its disadvantage is a substantial increase in object code size. The blowup factor varies greatly. Performance of `+b` and `+f` code is very similar on HP 98635A equipped systems.

The default compilation mode on Series 320 systems for C and FORTRAN is to generate inline 68881 instructions. By specifying `+M` you can force the compiler to generate library calls. This facility is provided to allow you to provide your own special purpose library routines. Either version of the code will execute correctly on Series 320, 330 and 350 systems. If you do not provide your own library routines, the calls will be satisfied automatically from *libc.a* and *libm.a*.

By default, compilation for C and FORTRAN on Series 330 and Series 350 systems is the same as on Series 320 and the `+M` option also works in a similar fashion. Series 330 and 350 systems also support the HP 98248A floating point accelerator which offers substantially greater performance than the 68881 coprocessor. By specifying `+ffpa` on the compile command line the compiler is directed to generate code for the accelerator. This code will not execute correctly unless the card is present at runtime. If you specify `+bfpa` the compiler will generate code to check for the presence of the accelerator at runtime and execute in the fastest mode available, much like the `+b` for the 68010 compiler. Code generated in this fashion will run on any Series 320, 330 or 350.

You should be prepared to accept slightly different levels of precision for the various floating point options on Series 300 since it is not possible to emulate hardware floating point accelerator support in software exactly or to expect different floating point hardware to give identical results in all cases. In all cases, however, precision is within the limits specified for the type specified by IEEE standards and the ranges shown in table 2-3.

All Series 300 systems actually contain two complete compile paths for C and FORTRAN. By default, the 68010 compiler will be accessed by the *cc()* driver program on systems equipped with 68010 processors (Series 310) and the 68020 compile path will be accessed on systems equipped with 68020 processors (all other Series 300). The `+x` and `+X` options enable you to cross compile for the other processor. For example, it would be possible to generate 68010 code that checks for the presence of the HP 98635A card at runtime on a Series 320 by using `+X +b` on the `f77` or *cc* command line.

Recommendations

Remember that the 68010 instruction set is a subset of the 68020 instruction set. Hence the 68010 compiler is frequently called the LCD (Least common denominator) compiler. By sacrificing some performance on 68020 systems you can generate one set of bits that will execute on all Series 300 systems. That is how all system code is generated.

On the other hand, if performance is more of a concern than object code compatibility with Series 310 systems, you should compile your code with the compiler that is appropriate for your processor (generally the default mode). This may require a separate executable version for Series 310 systems.

Series 310 floating point performance is improved substantially by the use of the HP 98635A floating point card. For example, the single precision *Linpack* program benchmarks at 6.9 Kflops on a system without the card. If the benchmark is compiled with `+b` and run on a system with the card, performance increases to 21.1 Kflops. Performance with the `+f` option is slightly higher still at 22.1 Kflops.

Series 330 and 350 systems also benefit with the use of optional floating point hardware. The same *Linpack* benchmark, when compiled on a Series 350 in the default mode (using the 68881 coprocessor), runs at 140.7 Kflops. It would actually slow down if code were generated for the HP 98635A floating point card. If the `+bfpa` option is added the performance increases to 422.8 Kflops. The `+ffpa` option further increases performance to 607.1 Kflops.

Benchmarks should properly be viewed with a great deal of cynicism. The only true benchmarks are your own programs. Obviously if your program does not use a great deal of floating point math or if the floating point math is not within critical sections, the performance enhancements will not be so dramatic.

While the `+b` option for the LCD compiler and the `+bfpa` option for the 68020 compiler do provide additional flexibility and performance, they also tend to increase the code size of statements involving floating point arithmetic. Note that they have no effect on other statements. The effect of code expansion varies so widely from one program to the next that it is difficult to give any accurate predictions. Since no source code modifications are needed in order to compile in any of the options discussed, it is recommended that a trial compilation of the actual program is the quickest and most accurate method to obtain code size information.

The *90% - 10% rule* definitely applies to floating point computation and the use of the `+b` or `+bfpa` options. Generally, around 90% of all computation time is spent within 10% of the executable code for a typical program. Therefore, if code size expansion is too painful when these options are used, it is often advantageous to compile only critical regions with these options and to compile the remainder of the program without any floating point hardware support. In this manner you get most of the advantages of floating point hardware performance without a great amount of code expansion. This method may require some reorganization of your file sets but the payoff is frequently well worth it.

Series 500 and 800 Floating Point

Series 500 does not require external floating point hardware support because the CPU provides it by default. There are, however, two versions of the CPU, *Focus-1* and *Focus-2* which have different floating point performance characteristics. *Focus-2* has much better floating point performance. Since the difference is strictly within the microcode interpretation, code generated on *Focus-1* is compatible on *Focus-2* and vice versa.

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

Porting VMS Code to HP-UX 2

This chapter presents guidelines that can help you port code from the VAX VMS(TM) 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 non-portable 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. Although it shares many source files with the C compiler, *lint* is a distinct program and it does not generate code.

All HP-UX FORTRAN compilers have the `-s` 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.

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

* 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 incorrect 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$**. The Series 300 FORTRAN compiler accepts procedure names with the **$** 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 in HP-UX can be obtained by using *open(2)* to return a file descriptor instead of SYS$ASSIGN's channel number and by using *ioctl(2)*, *read(2)*, and *write(2)* to set up character I/O and perform the I/O operations.

Graphics and Windows

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

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

* Tektronix is a trademark of the Tektronix Corporation.

HP-UX and VMS (along with several other vendors' systems) now share a common windowing system based on the *X Window System* from MIT. This brings an unprecedented level of 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 (most notably the use of structured data types). Consequently, an X application written in VMS FORTRAN will not just compile and run on HP-UX. HP provides some FORTRAN bindings for X that will make this task easier but source modifications will be necessary. X applications written in C, however, should be highly portable.

C language applications

It is easy to write nonportable code in C. Pointers, bit fields, structures and unions are all available to produce system dependent code that depends on byte ordering, alignment restrictions, and pointer representations. It is also easy to write highly portable code in C. See the language section below for a discussion of C language compatibility.

Since the C language provides no I/O capability, it depends on library routines supplied by the host system. Even on systems that are not based on the UNIX operating system, the programmer often finds a set of library routines that at least partially implement the set commonly found on implementations ofUNIX. Even when a high degree of compatibility exists, the same warnings about moving the data files apply. The data file 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. Once again, 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 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. The features change dramatically from one release to the next. This manual reflects only the Series 300 5.5 release, the Series 500 5.2 release, and the Series 800 1.1 release.

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. This extension is supported on all HP-UX implementations.
- <tab> characters. Tab characters encountered in columns 1-5 are treated differently between VMS and HP-UX. Tabs have the same behavior on VMS and all HP-UX implementations between columns 6-72
- Quotation mark ("), underscore (_), exclamation point (!), and percent sign (%). All HP-UX systems support these characters.

- Left and right angle brackets (< and >). These are used only to delimit variable expression within formats. These characters are supported only on Series 300.
- Ampersand (&) and dollar sign ($) are supported only on Series 300.
- <Control> L within source code for newpage is supported only on Series 300.
- Radix-50 character set. This character set is not supported on HP-UX.

Symbolic Names

- Symbolic names maximum length. VMS allows 31 characters within symbolic names. All are significant. All HP-UX implementations allow at least 255 characters within symbolic names. All are significant.
- Underscore in names is supported on all HP-UX implementations.
- Dollar sign in names is supported only on Series 300.

Data Types and Constant Syntaxes.

- The BYTE data type is supported on all HP-UX implementations.
- The LOGICAL*1, LOGICAL*2, LOGICAL*4 data types are supported on all HP-UX implementations.
- The INTEGER*2 and INTEGER*4 data types are supported on all HP-UX implementations.
- The REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types are supported on all HP-UX implementations.
- The DOUBLE COMPLEX data type (synonym for COMPLEX*16) is supported on all HP-UX implementations.
- Octal constants of the form `O'ddd'` are supported on all HP-UX implementations..
- Hexadecimal constants of the form `Z'ddd'` are supported on all HP-UX implementations.
- Hexadecimal constants of the form `'ddd'X` are supported only on Series 300.
- Octal, hexadecimal and Hollerith constants are considered to be "typeless" and may be used anywhere a decimal constant may be used. This feature is supported only on Series 300.

- Character constants have a maximum length of 2000 characters on VMS and on Series 300. Series 500 and 800 support unlimited length character constants.
- Hollerith is supported on all HP-UX implementations but in different ways. On the Series 300 and 800, Hollerith is treated internally as a synonym for a quoted character constant. On the Series 500, it is treated as a distinct data type.
- Octal constants of the form `'ddd'O` are supported only on Series 300.
- Octal constants of the form `"ddd` are not supported on HP-UX.
- REAL*16 is not supported on HP-UX.
- REAL*8 (D_floating) and COMPLEX*16 (D_floating) are not supported on HP-UX.
- RECORD and STRUCT data types are not supported on HP-UX.

General Statement Syntax and Source Program Format

- Exclamation point can be used for end of line comments on all HP-UX implementations.
- `D` is recognized in column 1 for debug lines on all HP-UX implementations.
- INCLUDE 'filename' is allowed for including source statements on all HP-UX implementations.
- Sequence numbering in columns 73-80. VMS ignores sequence numbers. HP-UX ignores anything in columns > 72.
- 99 continuation cards allowed on Series 300 and 800 only. Series 500 supports 20 continuation cards.
- A tab in columns 1-5 followed by a digit is interpreted as a continuation card. This feature is not supported on HP-UX.
- Up to 132 columns can be made significant under a VMS compiler option. HP-UX does not support this feature.
- DATA statements can be interspersed with specification statements only on Series 300.
- DATA statements can be interspersed with executable statements on all HP-UX implementations. However, the `-K` option must be specified on Series 300.

- Alternate forms of data type length specification are supported only on Series 300; for example:

```
INTEGER FOO*4
```

- Variables may be initialized when they are declared only on Series 300; for example:

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

- DATA statements may be used for initialization of common block variables outside of BLOCK DATA subprograms only on Series 300.
- Octal, hexadecimal and Hollerith constants are allowed within DATA statements on all HP-UX implementations.
- Octal, decimal, hexadecimal, and Hollerith constants may be used within DATA statements to initialize CHARACTER*1 variables only on Series 300.
- Two arithmetic operators may be consecutive if the second is a unary operator only on Series 300. 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.

Specification Statements

- IMPLICIT NONE turns off default type rules for variables on all HP-UX implementations.
- The VIRTUAL statement is supported only on Series 300.
- A limited number of the intrinsic functions are allowed to be used within the PARAMETER statement to define constants. These are ABS, CHAR, CMPLX, CONJG, DIM, DPROD, IAND, ICHAR, IEOR, IMAG, IOR, ISHFT, LGE, LGT, LLE, LLT, MAX, MIN, MOD, NINT, and NOT. Arguments to these functions must be constants in this context. These are supported only on Series 300.
- Support for the alternate form of the PARAMETER statement with different semantic connotations is found only on Series 300; for example:

```
PARAMETER ISTART = 3
```

- Symbolic constants may be used in run time formats only on Series 300.

- Symbolic constants may themselves be used to define COMPLEX symbolic constants within a PARAMETER statement on VMS but not on HP-UX.
- The VOLATILE statement is not supported on HP-UX.
- Multidimensional arrays may not be specified with only one subscript within EQUIVALENCE statements on HP-UX.
- The NOF77 interpretation of the EXTERNAL statement (non-ANSI semantics) is not supported on HP-UX.

Control Statements

- The DO ... WHILE control construct is supported on all HP-UX implementations.
- The DO ... END DO control construct is supported on all HP-UX implementations.
- Forcing FORTRAN66 semantics on DO loop evaluation by requiring a minimum of 1 iteration of the loop can be enabled via a compiler option on all HP-UX implementations.
- All HP-UX implementations allow jumps into IF blocks or ELSE blocks.
- Only Series 300 permits extended range DO loops. That is, allow 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.

Subprograms

- All HP-UX implementations allow all ENTRY names to be of types within the same type group but not necessarily of the same type; for example:

  ```
  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 on all HP-UX implementations.

- `%val`, and `%ref` are supported within actual argument lists only on the Series 300. `%descr` is not supported because *pass by descriptor* addressing is not used on the Series 300. Note that the **`$alias`** compiler directive provides the same functionality and it is supported on all HP-UX implementations.
- `%loc` is recognized as a built in function to compute the internal address of a datum only on Series 300.
- Only Series 300 recognizes 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. This feature is supported only on Series 300; for example:

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

  is equivalent to

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

- `&label` can be used in place of `*label` when specifying alternate returns only on Series 300.
- The controlling expression for an alternate return will be converted to INTEGER if necessary only on Series 300.
- Hollerith actual arguments are permitted on all HP-UX implementations. However, the corresponding formal parameter should be CHARACTER type on Series 300 and 800. There is no correctly corresponding formal parameter type on the Series 500.

Intrinsic Functions

- Mil-Std-1753 intrinsics ISHFT, ISHFTC, IBITS, BTEST, IBSET, and IBCLR are supported on all HP-UX implementations.
- The Mil-Std-1753 subroutine MVBITS is supported on all HP-UX implementations.
- ZEXT is supported only on Series 300 and Series 800.
- Transcendental intrinsics that take arguments in degrees (SIND, DSIND, COSD, DCOSD, TAND, DTAND, ASIND, DASIND, ACOSD, DACOSD, ATAND, DATAND, ATAN2D, and DATAN2D) are supported only on Series 300.

- The VMS specific intrinsics for INTEGER*2 and INTEGER*4 (AIMAX0, AIMIN0, AJMAX0, BITEST, BJTEST, CDABS, CDEXP, CDLOG, CDSQRT, DFLOAT, DFLOTI, DFLOTJ, DREAL, IIABS, IIAND, IIBITS, IIBSET, IIDIM, IIDNNT, IIEOR, IIFIX, IINT, IIOR, IISHFT, IISHFTC, IISIGN, IIXOR, IMAX0, IMAX1, IMIN0, IMIN1, ININT, INOT, JIABS, JIAND, JIBITS, JIDIM, JIDNNT, JIEOR, JINT, JIOR, JISHFT, JISHFTC, JISIGN, JIXOR, JMAX0, JMAX1, JMIN0, JMIN1, JMOD, JNINT, JNOT) are supported only on Series 300.
- VMS "system" support subprograms (DATE, EXIT, IDATE, TIME, SECNDS, RAN) are supported on Series 300 as a compiler option. These routines are not compatible with HP-UX system functions of the same name. ERRSNS is not supported on Series 300.
- VMS specific intrinsics to support the REAL*16 data type are not supported on HP-UX.

Input/Output Statements

- DECODE/ENCODE are supported on all HP-UX implementations.
- NAMELIST directed I/O is supported on all HP-UX implementations.
- Variable format expressions are supported only on Series 300.
- List directed internal I/O is supported only on Series 300.
- The TYPE statement is supported only on Series 300.
- An optional comma (,) is allowed to precede the iolist within a WRITE statement only on Series 300; for example:

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

  is equivalent to

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

- The UNIT and REC I/O specifiers will be converted to INTEGER if they are not already on Series 300 only.
- The RECL I/O specifier will not be converted to INTEGER if it is not already on HP-UX.
- 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 are supported on all HP-UX implementations.
- The `$` edit descriptor is supported on all HP-UX implementations.
- The `H` field descriptor can be used only with WRITE on HP-UX. VMS permits it to be used with READ as well.
- Default field descriptors are not supported on HP-UX.
- The `Q` edit descriptors is not supported on HP-UX.
- `$` and `ASCII NUL` carriage control characters are not supported on HP-UX.
- ACCEPT, DEFINE, DELETE, FIND, REWRITE, and UNLOCK statements are not supported on HP-UX.
- Key-field and key-of-reference specifiers are not supported on HP-UX.
- The VMS concept of indexed file access is not supported on HP-UX.
- Unsupported VMS keywords and I/O specifiers are flagged.
- Series 300 gives a nonfatal warning and ignores the keyword clause for the following VMS keywords:

  ```
  ASSOCIATEVARIABLE
  BLOCKSIZE
  BUFFERCOUNT
  CARRIAGECONTROL
  DEFAULTFILE
  DISP
  DISPOSE
  EXTENDSIZE
  INITIALSIZE
  KEYED
  MAXREC
  NOSPANBLOCKS
  ORGANIZATION
  READONLY
  RECORDSIZE
  RECORDTYPE
  SHARED
  TYPE
  USEROPEN
  ```

  S500 and S800 give an error.

- A comma (,) cannot be used to separate numeric input data to avoid having to blank fill (i.e. short field termination) on HP-UX.
- Extraneous parentheses are permitted around I/O lists for READ and WRITE statements on VMS. They are not supported on HP-UX; for example:

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

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 only on Series 300. They are implicitly converted to integer by truncation.
- Logical operands can appear in arithmetic expressions and numeric operands can appear in logical expressions only on Series 300.
- Noninteger array bound and subscript expressions will be converted to INTEGER by truncation only on Series 300.
- Character constants can be used in a numeric context; they are interpreted as Hollerith only on Series 300. Character constants and Hollerith are synonymous.

Miscellaneous

- Null strings are allowed in character assignments only on Series 300; for example:

```
c = '';
```

- `.XOR.` and `.NEQV.` are functionally equivalent operators only on Series 300 and Series 800.
- VMS represents `.false.` by 0 and `.true.` by 1. 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.

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 or when equivalencing the same storage locations with different data types. You should have few problems porting programs that require "large" local data storage onto HP-UX. On the Series 300, 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. The Series 500 does have a more restrictive size limitation on local data but it is possible to avoid this problem in most cases by using the **$EMA** directive which allows you to allocate large entities into separate virtual memory areas, and the **+Vc**, **+Vd**, and **+Vf** compiler options which cause all COMMON, SAVEd variables, and FORMAT strings into virtual memory respectively. See the *FORTRAN/77 Reference Supplement for HP9000 Series 500 Computers* for further details. The Series 800 is like the Series 300 in that data storage is limited only by the system configuration which is adjustable by the system administrator.

The more difficult 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, integral, floating point, Hollerith and character data types are not necessarily the same, however, so programs that depend on the internal representations of these data types will have to 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 three architectures in 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 on a Series 300 or Series 800 system, the compiler will report an error at compile time and refuse to generate further code. The Series 500 will not give a compile time error but the generated object code may be invalid.

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, the Series 300 has a **+A** compile line option and 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 `$HP1000 ALIGNMENT ON` is specified on a Series 800, the program will compile and produce the same results.

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 the Series 300 there are two tools that may be useful. First, you can use the cross referencing option to help look for uninitialized variables. Second, 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.

Series 500 supports both the `-K` option and cross referencing. Because of the architecture, variables on the Series 500 run time stack are initialized to 0 so the problem of uninitialized variables may not be apparent and the `-K` option may show nothing new.

Series 800 supports the `-K` option but does not yet support cross referencing.

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 in 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 a convenience to you, HP-UX FORTRAN does this 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:

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)[1]

[1] HP-UX places scratch files in the current directory unless the name includes an absolute path. File access permissions must be appropriate.

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 in HP-UX. In particular, all integral and floating point types have the same number of bits. Structs 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 2-3 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` varies between the two systems. HP-UX is generally less restrictive.
- The VMS C storage class specifiers `globaldef`, `globalref`, and `globalvalue` have no direct counterparts in HP-UX or other implementations of UNIX. Variables are local or global based strictly on scope or `static` class specifiers in HP-UX.
- The VMS C class modifiers `readonly` and `noshare` have no direct counterparts in 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 two tables (The first shows VMS C Floating Point Types; the second shows HP-UX 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

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 in HP-UX.
- VMS C identifiers are significant to the 31st character. HP-UX C identifiers are significant to at least 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. On the Series 500 all uses of the `register` reserved word are ignored simply because the architecture has no assignable registers. 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. Series 300 and 800 will report an error. Series 500, because the `register` reserved word has been ignored, will take no action.
- Type conversions on both systems follow the usual progression on implementations of UNIX. Both systems use conventions commonly referred to as *sign 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 HP-UX, which is not detected and produces an undefined result.
- 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, Series 300, and Series 500, but not on Series 800.

Preprocessor Features

- VMS supports an unlimited nesting of `#includes`. HP-UX guarantees only 8 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 in HP-UX.
- The following words are predefined in VMS but not in HP-UX: `vms`, `vax`, `vaxc`, `vax11c`, `vms_version`, `CC$gfloat`, `VMS`, `VAX`, `VAXC`, `VAX11C`, and `VMS_VERSION`.
- The following words are predefined in HP-UX but not in VMS: `hp9000s200` and `hp9000s300` on Series 300, `hp9000s500` on Series 500, and `hp9000s800` on Series 800. `hpux` and `unix` are predefined 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$LIBRARY: VAXCURSE.OLB`, `SYS$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 except on the Series 500 which supports only `end`.

The Pascal Language

No information is currently available on VMS Pascal.

Porting from BSD4.3 to HP-UX 3

No information is currently available on BSD4.3.

Porting Across HP-UX 4

While HP-UX is highly standardized, it is not a homogeneous environment. 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 the Series 300, 500 and 800 systems.

Interlanguage communication relates to porting across architectures. FORTRAN and Pascal programs must deal with this problem 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.

A second constraint on interlanguage communication is that input/output operations are best performed in the language in which the main program is written because each language has certain startup code that is specific to the implementation language of the main program that does input/output initialization. Methods exist to do input/output from external routines, but they lack generality. 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. Hence it is recommended that all such operations 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.

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

This table shows the sizes of the C data types on the three architectures:

Table 4-1. C Data Types

Type	Size	Alignment (300)	Alignment (500)	Alignment (800)
char	8 bits	byte	byte	byte
short	16 bits	2 byte	2 byte	2 byte
int	32 bits	4 byte[1/2]	4 byte	4 byte
long	32 bits	4 byte[1/2]	4 byte	4 byte
float	32 bits	4 byte[1/2]	4 byte	4 byte
double	64 bits	4 byte[1/2]	4 byte	8 byte
pointer	32 bits	4 byte[1/2]	4 byte	4 byte
struct/union		4 byte[1/2]	2 or 4 byte[3]	2,4 or 8 byte[3]

[1] Within a struct or union the alignment is 2 byte.
[2] Prior to release 5.15 the alignment is 2 byte.
[3] 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, will reduce execution time. Using this type should not hinder portability, however, its usefulness on other systems will vary, since some ignore it. In particular, the Series 500 ignores all uses of the `register` specification because it is a stack based system that has no registers. 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 seven for external identifiers (identifiers that are defined in another source file) is 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 in HP-UX: `hp9000s200` and `hp9000s300` on Series 300, `hp9000s500` on Series 500, and `hp9000s800` on Series 800. `hpux` and `unix` are predefined on all HP-UX systems.

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 constant. 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 and 500 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`.

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.

Series 300 and 800 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. Series 500 will not report the error; the overflow is silent.

Structure Assignment and Functions

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

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

Multiple structure assignments such as `s1=s2=s3;` are **not** supported on the Series 500. Structure valued functions support storing the result in a structure:

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

The result of a structure-valued function is not an **lvalue** (i.e. it cannot be used on the left hand side of an assignment statement by itself), and a direct dereference of field from it is not legal. For example, the following statement will yield a compiler syntax error:

```
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. In the HP-UX implementation of C, referencing a null pointer normally causes a run time error on the Series 500 but if you try to read using a null pointer on the Series 300 or 800, a value of zero is returned. The Series 500 supports the `-Z` compiler option which causes null pointers, when dereferenced, to return 0. 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.

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 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 500 and 800, parameter lists are stacked towards decreasing addresses (though the stack itself grows towards higher addresses). To step through a parameter list, decrement the pointer as shown in the following 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 will print its three parameters in order. Note that the only difference between this function and the similar one for Series 300 HP-UX is that `ptr` is decremented instead of incremented.

For portability, it is 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.

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.

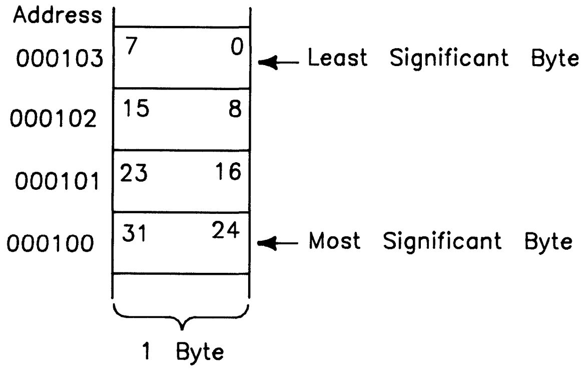

Figure 4-1. Memory Organization

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)`. It is good programming practice to use parentheses with all expressions so you know the order of evaluation. Since there is no C standard for order of evaluation of expressions, avoid 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

Due to the Series 500 hardware `auto` variables are implicitly initialized to 0. This is not the case on Series 300 and 800, and is most likely not the case on any other implementation 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 *sign 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. conversions of `unsigned char` go first to `char` and then to `int` before they are used in an expression).

The following program will print:

```
Sign 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("Sign preserving\n");
if (ui < i)
 printf("Unsigned comparisons performed\n");
  }
```

Code/Data Limitations

The following limitations exist on the Series 500 C compiler:

- A maximum of 2^19 bytes of local variables in any procedure.
- A maximum of 2^19 bytes of parameters in any function call.
- Any branch instruction generated by a procedure must be within 2^18 bytes of its target.
- Structure functions cannot return a structure bigger than 2^24 bytes.

If any of the above limitations are violated, you get the message `impossible reach` from the assembly step of *cc*. Other limitations are:

- A maximum of 65 535 lines of assembly code per file generated by *cc*. If you exceed this, you get a `too many lines` message from the assembler. To work around this, break your program into smaller pieces.
- A maximum of 2^19 bytes of global scalar data (includes all global or static scalar variables, all global and static structures, and 4 bytes for each global or static array). If this is exceeded, you get a `byte offset too large` message from the linker, *ld*.

Series 300 and 800 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.

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
-p	Enable profiling.	Not supported on Series 500.
-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.
-w	Suppress warning messages.	Not supported on either Series 300 or 500.
-Z	Allow dereferencing of null pointers.	Has no effect on Series 300 pointers.
-z	Allow run time detection of null pointers.	Not supported on Series 300. Has no effect on Series 500.

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 a 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 on the following page).

Table 4-3. C Interfacing Compatibility

C	Pascal	FORTRAN
char	none	byte
unsigned char	char	character[1]
char* (string)	none	none
unsigned char* (string)	PAC+chr(0)[2]	Array of char+char(0)
short (int)	-32768..32767	integer*2
unsigned short (int)	none[4]	none
int	integer	integer (*4)
long (int)	integer	integer (*4)
unsigned (int)	none[3]	none
float	real	real (*4)
double	longreal	real*8
type* (pointer)	^var, pass by reference, or use **anyvar**	none
&var (address)	addr(var)[5]	none
*var (deref)	var^	none
struct	record[6]	[7]
union	record case of...	equivalence[8]

[1] Care must be taken in calling FORTRAN from C (or Pascal), since the character could reside on an odd boundary, thus causing a memory fault.

[2] PAC = **packed array[1..*n*] of char**. Note that for assignment compatibility with string literals, and for **STRMOVE** to work, a PAC must have the range 1..*n*.

[3] 0..**4294967284** is not allowed, since 2 147 483 647 is **MAXINT**.

[4] 0..**65535** will not generate a 16-bit value unless it is in a packed structure.

[5] Requires **$SYSPROG$** capabilities.

[6] This cannot always be done, since C and Pascal use different packing algorithms.

[7] This can be simulated with multiple equivalenced arrays, but it is extremely tedious.

[8] Actually, a union is intended to represent different objects at different times. Thus the appropriate interface can be set up to match the type for the given context.

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 /a/array(1000,10)
   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, 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 in Figure 4-2 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 */
  }
```

Figure 4-2. Source for Main C Program

Figure 4-3 shows the source 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.
```

Figure 4-3. Source for the Pascal module (`pfunc.p`)

The command line for producing the Pascal relocatable object is

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

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

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

Which produces the following output:

```
2147483647
-2147483648
```

Unfortunately, calling Pascal functions from C on the Series 500 is different than on the Series 300. The following page has an example.

The examples shown in Figures 4-4 and 4-5 show the code for calling a Pascal function from a C program on Series 500; first the C code:

```
main ()
/*  CALL A PASCAL FUNCTION */
{
      int     a, b, psubs_changenum();

      a = -4;
      b = 3;
      printf ("Before the call, a = %d\n", a);
      psubs_changenum (&a, b); /* NOTE THE USE OF & FOR VAR PARAM */
      printf ("After the call, a = %d\n", a);
}
```

Figure 4-4. Source for Main C Program (`main.c`)

You now see the Pascal code:

```
 module psubs;

 export
        function changenum (var num1: integer; num2 : integer): integer;

implement

function changenum (var num1: integer; num2 : integer): integer;
{ FUNCTION TO ADD NUM1 TO NUM2. IF THE NEW VALUE OF NUM1 IS
  NEGATIVE THEN 0 IS RETURNED, OTHERWISE 1 IS RETURNED.     }

 begin
        num1:= num1 + num2;
        if num1 < 1 then
            changenum:= 0
        else
            changenum:= 1
end; { CHANGENUM }

end. { MODULE PSUBS }
```

Figure 4-5. Pascal Function Source (`psubs.p`)

The commands to compile these files into the executable file named `a.out` are:

```
pc -c psubs.p
cc main.c psubs.o -lpc
```

The `-lpc` tells the linker to link any required object files from the Pascal library.

Note that the C program refers to the Pascal function as `psubs_changenum`. This follows the general form:

<module name>_<function name>

If you want to access I/O functions from a Pascal function within a module, you must declare the files you wish to use (including `input` and `output`) in the function.

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

This table shows the sizes of the FORTRAN data types on the three architectures:

Table 4-4. FORTRAN Data Types

Type	Size	Alignment (300)	Alignment (500)	Alignment (800)
`character`	8 bits	2 byte	2 byte	2 byte
`Hollerith`[2/5]	8 bits	2 byte	2 byte	2 byte
`byte,logical*1`[2/3/4]	8 bits	2 byte	2 byte	2 byte
`logical*2`[2/3]	16 bits	2 byte	2 byte	2 byte
`integer*2`[2/3]	16 bits	2 byte	2 byte	2 byte
`logical (*4)`[3]	32 bits	4 byte[6]	4 byte	
`integer (*4)`[3]	32 bits	4 byte[6]	4 byte	4 byte[7]
`real (*4)`[3]	32 bits	4 byte[6]	4 byte	4 byte[7]
`double precision,real*8`[3/4]	64 bits	4 byte[6]	4 byte	8 byte[7]
`complex (*8)`[3]	64 bits	4 byte[6]	4 byte	4 byte[7]
`double complex,complex*16`[2/3/4]	128 bits	4 byte[6]	4 byte	4 byte[7]

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. 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.

1 8 bits per character. No **null** terminator.
2 This type is an extension to ANSI FORTRAN77.
3 ANSI does not support a length descriptor " *n ".
4 Synonymous types.
5 Character and Hollerith are synonymous on S300 but distinct types on S500. Hollerith on S500 has no corollary in C and Pascal.
6 The minimum alignment can be reduced to 2 byte if the `+A` compiler option is specified.
7 The minimum alignment can be reduced to 2 byte if the `$HP1000 ALIGNMENT ON` inline option is specified.

Error Conditions

The various HP-UX FORTRAN compilers are based on different compiler technologies. In general, therefore, it is not possible to detect compile time error conditions in the same ways on each system. Some errors are detected at compile time on a Series 300, for example, that might be detected at link time on a Series 500. This implies that compile time error messages are not compatible between the systems. Series 300 attempts to give plain text error messages. Series 500 and 800 give error numbers with optional plain text messages in a different format.

Run time errors are much more compatible because all three 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.

Series 300 is more relaxed about statement sequencing than is required by ANSI. In many cases, duplicate declarations are not disallowed although if they are conflicting the result may be undefined. On the other hand, Series 500 and 800 will generally enforce ANSI requirements more strictly.

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.

Lower Case Not Supported

Lower case letters are not supported in ANSI FORTRAN 77. If `$OPTION ANSI ON` is specified you will get a non-fatal warning for each ANSI violation to include a warning for each lower case letter. The resulting `stderr` file can become large for programs written in lower case.

String Constants

String constants are limited to 2000 characters in length on Series 300 whereas they are essentially unlimited on Series 500 and 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 500 and 800 make 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 since data types are compatible on all HP-UX systems, unformatted data files created on an HP-UX system can be read on any other HP-UX system. The ability to read unformatted data files across system boundaries is particularly useful since unformatted I/O is typically the fastest data storage and retrieval mode available.

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

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

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

Here is the **reader program**.

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

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

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

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

Optimization

FORTRAN is particularly well suited for compiler optimization. All HP-UX compilers will perform those optimizations that are most effective on the particular architecture of the system. However, you cannot assume that the same optimizations will be performed on all HP-UX systems.

Parameter Passing

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.

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 so it will be flagged as a nonfatal error on that system.

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 three systems. Options that are the same on all three 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 data alignment	S300 only. S800 uses an inline directive.
+b	Floating point option	S300 only
+bfpa	Floating point option	S300 only.
+E0	Enable subset of +e features	S300 only
+E1	Enable subset of +e features	S300 only
+e	Enable FORTRAN66 features	S300 only
+e	Write errors to stderr	S500 only
+f	Floating point option	S300 only
+ffpa	Floating point option	S300 only
+F	Enable program analysis	S500 only
-G	Berkeley style profiling	S300 and S800 only
-K	force static allocation	S300 has side effects
+M	Floating point option	S300 only
+N	Adjust table sizes	S300 only
-O	Assembly code optimizer	S300 and S800 only
-p	Prepare for profiling	S300 and S800 only
+Q	Specify option file	S500 and S800 only
-S	Compile; don't assemble	S300 and S800 only
+T	Procedure traceback	S500 and S800 only
-u	Implicit typing off	Can be overridden in a program unit on S300
+U[1]	Case is significant	S300 only
-Vc	Virtual COMMONs	S500 only
-Vd	Virtual SAVEs and DATA	S500 only
-Vf	Virtual FORMATs	S500 only
-w66	Suppress FORTRAN 66 warnings	S300 only
+X	68010 code generation	S300 only
+x	68020 code generation	S300 only
-Y	enable 16-bit NLS	optional language parameter (300)

1 Do not confuse this with -U which is supported on all HP-UX implementations.

Compiler Directives

`ALIAS`: The meaning of the ALIAS directive differs slightly between the Series 300 and other HP-UX implementations. On all systems, `ALIAS` allows you to specify the parameter passing mechanism to be used when calling the aliased procedure. Series 300 FORTRAN does not allow the `%desc` passing mode because this mechanism is not used on Series 300.

`HP1000 ALIGNMENT ON` is supported only on Series 800. This option causes alignment restrictions to be reduced at the expense of performance. Its effect is comparable to the `+A` Series 300 compiler directive. See Chapter 2 for a more detailed discussion of its characteristics.

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.*

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`[2/5]		
`byte`, `logical*1`[2/3/4]	-128..127 or `boolean`	`char`[1]
`logical*2`[2/3]	-32768..32767	`short`
`integer*2`[2/3]	-32768..32767	`short`
`logical (*4)`[3]	`integer`	`long` or `int`
`integer (*4)`[3]	`integer`	`long` or `int`
`real (*4)`[3]	`real`	`float`
`double precision`, `real*8`[3/4]	`longreal`	`double`
`complex (*8)`[3]	`record`	`struct`
`double complex`, `complex*16`[2/3/4]	`record`	`struct`

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.

1 FORTRAN can pass its characters to Pascal and C, but care must be taken when calling in the other direction. The character type may reside on an odd boundary, thereby causing a memory fault.
2 This type is an extension to ANSI FORTRAN77.
3 ANSI does not support a length descriptor `*n`.
4 Synonymous types.
5 Character and Hollerith are synonymous on Series 300 and Series 800 but distinct types on Series 500. Hollerith on Series 500 has no corollary in C and Pascal.

FORTRAN cannot easily emulate Pascal `records`, nor C `structs`. This type of interface must be done via multiple equivalenced arrays and can be quite tedious.

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

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 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);
   }
```

C Function (`chcount.c`)

```
   $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)
      INTEGER I, STATUS

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

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

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, 500 and 800 which are covered in this section.

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

Table 4-7 shows the sizes of the Pascal data types on the three architectures:

Table 4-7. Pascal Data Types

Type	Size	Alignment (300)	Alignment (500)	Alignment (800)
`char`	8 bits	1 byte	1 byte	1 byte
`boolean`	8 bits	1 byte	1 byte	1 byte
`shortint`	16 bits	[1]	[1]	2 byte
subrange of `integer`	16 bits[2/3]	2 byte	2 byte	[4]
`integer`	32 bits	4 byte[5]	4 byte	4 byte
`longint`	64 bits	[1]	[1]	4 byte
`enumeration`	16 bits[3]	2 byte	2 byte	[4]
subrange of `enumeration`	16 bits[3]	2 byte	2 byte	[4]
`real`	32 bits	4 byte[5]	4 byte	4 byte
`longreal`	64 bits	4 byte[5]	4 byte	8 byte
pointer	32 bits	4 byte[5]	4 byte	4 byte
`set`	[6]	[6]	[6]	[6]

1 This predefined type is not supported on Series 300 or Series 500.
2 On Series 300 and 500, allocation is 16 bits if the range is between -32768 and 32767. Otherwise it is 32 bits.

3 On Series 800, allocation can be 8, 16 or 32 bits based on the declared size. Refer to the HP Pascal Programmer's Guide for details.
4 On Series 800, alignment can be byte, 2 byte or 4 byte based on the declared size. Refer to the HP Pascal Programmer's Guide for details.
5 On Series 300, alignment can be reduced to 2 byte if +A is specified.
6 Size and alignment vary. See the appropriate Language Reference.

Command Line Options

Table 4-8 shows some differences in the Pascal compiler (*pc*) command between the three 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
`+a`	generate `.a` files instead of `.o` files	Series 300 only
`+E`	link with `/lib/libpcesc.a`	Series 500 only
`+F`	produce program analysis information	Series 500 only
`-G`	produce information for *gprof(1)*	Series 300 only
`+H`	display/set maximum heap size	Series 500 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
`-P`	set lines per listing page	Series 300 and 500 only
`+Q`	read a file of compiler options	Series 500 only
`+U`	external names made upper case	Series 500 only
`+W`	display/set working set size	Series 500 only
`-w`	suppress warning messages	Series 300 and 500 only
`+X`	generate 68010 code	Series 300 only
`+x`	generate 68020 code	Series 300 only
`-Y`	enable HP15 support in strings	Series 300 and 500 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	Available on all HP-UX implementations. Series 500 adds an underscore as a prefix to the alias name whereas Series 300 and 800 leave the name unaltered.
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.
AUTOPAGE	Series 500 only. Controls pagination of listing.
BUILDINT	Series 800 only. Causes the compiler to build an intrinsic file rather than an object code file.
CALL_PRIVILEGE	Series 800 only. Specifies lowest privilege allowable for calling function or procedure.
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	Available on Series 300 and 500 only. Causes line number debugging information to be included in the relocatable file.
EXEC_PRIVILEGE	Series 800 only. Specifies level of execution of this function or procedure.
EXTERNAL	Series 800 only. Used in conjunction with the GLOBAL option, enables you to compile one program as two or more compilation units.
EXTNADDR	Series 800 only. Specifies that a pointer has an extended address.
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*.
HP15	Series 500 only. Permits HP15 characters within string literals and comments. Series 300 has a corresponding NLS_SOURCE directive.
IDSIZE	Series 500 only. Specifies number of significant characters in identifiers.
IF	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.
LINE_INFO	Series 500 only. Listing option.
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 500 and 800.
LINESIZE	Series 500 only. Controls line buffering for TEXT files.
LIST_CODE	Series 800 only. When LIST is also on, a mnemonic listing of object code is produced.
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 which file the object code is to be written to.
NLS_SOURCE	Series 300 only. Allows HP15 characters within string literals and comments. The Series 500 has a corresponding HP15 directive.
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.
OVFLCHECK	Series 300 and 800 only. Switches overflow checking on or off. Series 300 does this on a statement by statement basis whereas Series 800 does it for an entire routine.
PAGEWIDTH	Series 300 only. Controls width of listing.
POP	Series 800 only. Pops the compiler option stack.
PUSH	Series 800 only. Pushes a compiler option onto the option stack.

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.
SAVE_CONST	Series 300 only. Controls scope of structured constants.
SEARCH	Available on all HP-UX implementations. The Series 300 syntax requires a list of quoted file names. The Series 500 syntax uses a single string with a list of module names.The Series 800 requires a quoted list of files created by MLIBRARY.
SEARCH_SIZE	Series 300 only. Changes number of external files that can be searched. The default is 9.
SET	Series 800 only. Used with the IF option to associate boolean values with identifiers.
SKIP_TEXT	Series 500 only. Skips source text.
SPLINTR	Series 800 only. Specifies what intrinsic file is to be read.
STANDARD_LEVEL	Series 500 only. Sets level of extensions that can be used without triggering a warning. Series 300 uses ANSI and SYSPROG.
STATS	Series 500 only. Display compiler options.
STRINGTEMPLIMIT	Series 800 only. Specifies the maximum size of a temporary string used within a string expression.
SUBPROGRAM	Series 500 and 800 only. Separate compilation facility. Use modules instead.
SUBTITLE	Series 500 only. Prints a listing subtitle.
SYSINTR	Series 800 only. Specifies the intrinsic file to be searched for information on intrinsic procedures and functions.
SYSPROG	Series 300 only. Allows use of system programming extensions. Equivalent to $STANDARD_LEVEL 'hp_MODCAL'$ on the Series 500.

TABLES	Available on all HP-UX implementations. Series 300 forbids its use within a procedure body whereas Series 500 and 800 permit it anywhere.
TMPSTRMAX	Series 300 only. Specifies maximum temporary storage for long string operations. Series 800 uses STRINGTEMPLIMIT.
TITLE	Series 500 and 800 only. Prints a listing title.
TYPE_COERCION	Series 500 and 800 only. Relax type checking.
UPPERCASE	Series 800 only. All external names are shifted to upper case ASCII.
UPSHIFT_LEVEL1	Series 500 only. Makes all external names upper case.
VERSION	Series 800 only. Specifies a version stamp to be placed in the object file.
VISIBLE	Series 500 only. Specifies default entry points.
WARN	Series 300 only. Causes compiler warnings to be suppressed. It has no effect on fatal errors.
WIDTH	Series 500 and 800 only. Sets the accepted width of a source line. Anything beyond this width is ignored.
XREF	Series 800 only. Causes a cross reference map to be produced for each compilation block.

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, 500 and 800 HP-UX Pascal.

Program Structure

- Modules are supported on all HP-UX implementations but some syntactic and semantic differences exist. For example, Series 500 and 800 require 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 500 and 800 provide other mechanisms as well.
- Absolute addressing is supported on the Series 300 only.
- Only Series 500 supports nested comments.

Types

- Series 500 and 800 Pascal has both procedure and function types, while Series 300 has only procedure types (and hence no `fcall()`). Assignments to procedure variables have a different syntax.
- Maximum string length in Series 500 Pascal is 32 767 characters. On the Series 300, it 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.
- Only Series 500 and 800 support type coercion.
- All HP-UX implementations support constant expressions but different restrictions apply on operators. Series 300 does not recognize `abs()` or `odd()` within a constant expression. Series 500 will not allow Unary `+` or `-` operators. Series 300 does not allow **real** expressions within the CONST section.
- 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.

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.
- Series 300 allows the use of a label on a RECOVER statement. Series 800 does not.

I/O

- I/O of enumerated types is supported on the Series 300 and 800 only.
- 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 **associate()** and **reset()** procedures.
- Series 800 uses the *options string* parameter on **reset()**, **rewrite()**, **open()**, and **append()** procedures. Series 300 and 500 ignore this parameter.
- Output to **stdout** is line buffered on the Series 800.

Miscellaneous

- Series 800 supports **readonly** parameters. Series 300 and 500 do not.
- Series 300 Pascal allows using a file variable as a parameter to the **sizeof** function; Series 500 and 800 do not.
- Only Series 800 implements the **bitsizeof()** function.
- All HP-UX implementations support symbolic debugging. Series 300 and 500 support *cdb*. Series 800 supports *xdb*. Debugger syntaxes differ.
- Only Series 300 allows **ord()** on a pointer type.
- Only Series 300 allows **addr()** of a constant.

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`[1]
`char`	`char`	`character*1`[2]
`integer`	`long; int`	`integer` (*4)[5]
-32768..32767	`short`	`integer` (*2)[5]
`real`	`float`	`real` (*4)[5]
`longreal`	`double`	`double precision`
enumerated type	use **`short`**	use **`integer*2`**[5]
subrange (32-bit)	use **`long; int`**	use **`integer*4`**[5]
subrange (16-bit)	use **`short`**	use **`integer*2`**[5]
`set`	none	none
`record`	`struct`[3]	[4]
^<*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.

[1] Passing a boolean to a FORTRAN **logical*2** or **logical*4** won't work because if it is on an odd boundary you get a memory fault, and it is if on an even boundary, the FORTRAN **logical*2** will test the LSB of the corresponding odd byte.

[2] Care must be taken that the Pascal **char** does not lie on an odd byte. The FORTRAN type **character*1** assumes that the character is in the even byte of a 16-bit word.

[3] Care must be taken to make sure that the fields align. C packs its **struct**s very tightly.

[4] Can be simulated with multiple equivalenced records.

[5] integer*2, integer*4, real*4 are extensions to ANSI standard FORTRAN77. Standard data types in FORTRAN77: character, integer, real, logical, double precision, complex.

Here are a few recommendations for Pascal programs that must interface to FORTRAN and/or C. These suggestions apply to the variables that will interface with the other languages, and not to all variables in the program.

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

To call an external (FORTRAN, C, or assembler) procedure, 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.

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 structures 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. 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.

The following example shows how to call a user defined C function from a Pascal program. You first see the Pascal source, then the C 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.
```

The following page has 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
```

System Calls and Subroutines 5

This chapter explains differences between system calls and subroutines among the HP-UX implementations. HP-UX may not support the same set of system calls and subroutines if you port from another implementation of UNIX. No semantic differences between HP-UX and UNIX routines of the same name were documented. Substantial differences probably do not exist.

The first part of the chapter lists system calls that have differences; the second part lists subroutines. If you have hardware dependencies or need more information about a routine, see the *HP-UX Reference* for your system.

The information in this chapter was accurate for the versions of HP-UX mentioned earlier. Updates and improvements to the HP-UX system may invalidate some entries in this section. The *HP-UX Reference* for your version is the final word on what routines are available on a particular system.

System Calls

acct	Many system differences exist. See the *HP-UX Reference.*
brk	Many system differences exist. See the *HP-UX Reference.* Reference under *brk*(2).
dup2	Not available on Series 500.
ems	Available on Series 500 only.
errinfo	Available on Series 500 only.
errno	One additional *errno* value is implemented on the Series 500 (EUNEXPECT). Series 500 also has slightly different definitions of ENOMEM and EMLINK.
exec	The Each HP-UX system has different object module formats that may affect use of *exec.*
exit	Job control is supported only on Series 800.
fchmod	Not available on Series 500.
fchown	Not available on Series 500.
fcntl	Series 500 does not support F_GETLK, F_SETLK, and F_SETLKW.
fork	*Fork* will fail on the Series 300 if there is not enough swapping memory to create the new process (ENOSPC). On the Series 500 it will fail if there is not enough physical memory to create the new process (ENOMEM). *profil* is not supported on Series 500.
ftime	Not available on Series 500.
getgroups	Not available on Series 500.
getitimer, setitimer	On Series 500 an error is generated if a call is made to *getitimer/setitimer* in the [vfork, exec] window.
getpgrp2	Series 800 only.
getprivgrp, setprivgrp	Not available on Series 500.
ioctl	Asynchronous I/O is supported only on Series 800.
kill	Job control is supported only on Series 800.

link	On the Series 500, for Structured Directory Format (SDF) discs, if **path2** is "..", then that directory's i-node will be altered such that its ".." entry points to the directory specified by **path1**.
lockf	On both Series 300 and 800, EINVAL errors are defined differently.
memadvise, memallc, memfree, memchmd, memlck, memulck, memvary	Series 500 only.
mknod	On both the Series 500 and 300 there is an additional value — 0110000 — available under file type that specifies network special files.
open	On the Series 500, demand loaded program files that are not shared remain open until all of the code and data have been loaded. Also, execute and write access are mutually exclusive.
plock	On both the Series 300 and 500 the call to *plock* is not allowed in the [vfork,exec] window.
profil	Not available on the Series 500. The sampling frequency is 50 Hz. on the Series 300.
ptrace	Much of the functionality of *ptrace* is dependent on the underlying hardware. An application which uses this intrinsic should not be expected to be portable across architectures or implementations. Not available to user on Series 500.
read	Series 500 does not support **nbyte** values greater than 512K when **fildes** is associated with a device file. See the *HP-UX Reference* for exceptions.
readv, writev	Not supported on Series 500.
reboot	Series 300 and 800 only. Many differences exist. See the *HP-UX Reference.*
rmdir	On the Series 500 the directory identifiers "." and ".." are not recognized by *rmdir.*

rtprio	On Series 500 there are some suggestions for the use of *rtprio*. See the *HP-UX Reference* for details.
setgroups	Not available on Series 500.
setpgrp2	Not available on Series 300 or 500.
select	Many system differences exist. See the *HP-UX Reference*.
shmctl	Series 300 has differences in the handling of EACCESS.
shmget	On Series 500, Shared memory segments larger than 16384 bytes are paged virtual segments; otherwise they are non-paged virtual segments.
shmop	There are extensive implementation differences between the Series 300, 500 and 800 involving *shmaddr* and various variables and constants. See the *HP-UX Reference* for details.
signal	There are extensive implementation differences between the Series 300, 500 and 800. See the *HP-UX Reference* for details.
sigspace	On the Series 500 **sigspace** is ignored as a no-op. The return value is always 0. On the Series 300 the guaranteed space is allocated with *malloc* and may interfere with other heap management mechanisms.
sigvector	The SV_BSDSIG flag is not supported on Series 300 or 500.
stat	Series 500 has special meanings for **st_size**. It also does not support **st_netdev** and **st_netino** fields.
swapon	Series 300 and 800 only.
times	For Series 500 computers with multiple CPUs, the child CPU times listed can be greater than the actual elapsed real time, since the CPU time is counted on a per-CPU basis.
trapno	Series 500 only.
uname	the **U** version field is not supported on Series 300 or 500.
unlink	On the Series 500 the last link to a directory cannot be unlinked if the directory is not empty.

ustat	On the Series 300, **f_tfree** and **f_flksize** are reported in fragment size units. On the Series 500, **f_fname[6]** is the driver name, not the file system name.
vfork	Any program which relies upon the differences between *fork* and *vfork* is not portable across HP-UX systems. On Series 300 and 800, a call to *signal()* in the [*vfork, exec*] window which is used to catch a signal can affect handling of the signal by the parent. See the *HP-UX Reference* for further details.
vsadv	Implemented only on the Series 500.
vson, vsoff	Implemented only on the Series 500.
wait3	Implemented only on the Series 800.
write	The Series 500 has some anomalies that are listed in the *HP-UX reference.* The size of a pipe (NPIPE) is 5120 bytes on the Series 500 and 8192 bytes on the Series 300.

Subroutines

abs	On HP-UX, calling *abs* with the most negative number returns that number.
atoi, atol	On the Series 300 and 500 these two subroutines are identical to each other.
catgetms	Series 800 only. Use *getmsg(3C)* on Series 300 and 500.
clock	clock resolution is 20 milliseconds on the Series 300, the default is 10 milliseconds on the Series 500.
ctime	tztab is not supported on the Series 300 or 500.
ecvt	Series 300 and 500 use *nl_gcvt()* to provide NLS radix support.
end	The following items pertain to Series 500 HP-UX: etext and edata are not supported; memallc is more efficient than malloc for setting the program break
gpio_get_status	On the Series 300 and 500, *x* is 2 for the current GPIO card.
gpio_set_ctl	On the Series 300 and 500, *x* is 2 for the current GPIO card.
hpib_abort	On the Series 300, the HP 98625A/B HP-IB interface does not clear the ATN line.
hpib_bus_status, hpib_card_ppoll_resp, hpib_rqst_srvce, hpib_send_cmd	Many system differences exist. Refer to the *HP-UX Reference.*
hpib_status_wait, hpib_wait_on_ppoll	On the Series 500, when either of these subroutines is in progress all other bus activity is held off until the subroutine has completed. It is recommended that a timeout be in effect before the subroutine is called.
io_eol_ctl, io_get_term_reason, io_interrupt_ctl, io_on_interrupt, io_speed_ctl, io_timeout_ctl, io_width_ctl	Many Series 300, 500 and 800 hardware dependencies. See the *HP-UX Reference.*

nl_init	Series 800 only. Series 300 and 500 use routines described in *nl_ctype(3C)* instead.
nl_langinfo	Series 800 only. Use *langinfo* instead on Series 300 and 500.
nl_printf, nl_fprintf	Series 800 only. Series 300 and 500 use *printmsg* and *fprintmsg* instead.
nl_scanf, nl_fscanf	Series 800 only.
nl_strcmp, nl_strncmp	Series 800 only. Series 300 and 500 use routines in *nl_string(3C)* instead.
nlist	Not implemented on the Series 500.
perror	The Series 500 provides the additional error indicator *errinfo*.
setbuf	On the Series 500 the system call *memallc* is used instead of *malloc*.
string	On the Series 300 the argument *N* is limited by the process size; on the 500, it is limited to about 500 Mbytes.
trig	The approximate limit for the values passed to these functions is 2.98E8 for *sin* and *cos*, 1.49E8 for *tan*, 1.29E4 for *fsin* and *fcos*, and 6.43E3 for *ftan*.

Pascal Workstation to HP-UX 6

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 500 or 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 has been done so HP-UX Series 300 Pascal is compatible with HP-UX Series 500 Pascal. Note that this is different from the 2.1 release of HP-UX Series 300 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 in 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 characteristics. 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.

```
..m off on
  $sysprog$
  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 (disc drivers, keyboard, etc.). Here we will look at some of these same capabilities on HP-UX.

Graphics

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 (old HP1000 FORTRAN only allowed five-character names; thus, there are procedure names like `ZDPMM`, `ZIPST`, `ZLSTL`, `ZPGDI`, etc.); as well as a Pascal interface (i.e., Pascal integers and strings).

On HP-UX, for expediency, the original HP1000 FORTRAN DGL library was ported. This means that the names have reverted to those shown above. Also the parameters are all passed by reference (`var`); and strings are not Pascal strings, but FORTRAN character arrays (with a separate length parameter), and 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.

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.

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.

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

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[1]/[2]. Note that in HP-UX the external names are referenced via 32-bit addresses, and thus 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 psuedo-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.

---

1 **atrans** converts Workstation assembly code into Series 300 HP-UX assembly code using the assembly syntax available since release 5.15.

2 Do not confuse this command with **astrn()**, which translates S200 assembly code from pre 5.15 releases to S300 assembly code post 5.15 release.

Accessing Series 300 Shared Memory 7

A frequent problem Series 300 FORTRAN users encounter is the difficulty in using shared memory between cooperating processes. The following example gives one possible solution. It is not to be construed as a portable solution to Series 500 and 800. However, it is an case study of interlanguage communication.

Assume that two FORTRAN programs must share an amount of shared memory. One way to make this possible is to consider this memory to be a block of COMMON storage. This memory can be *defined* by means of a separate assembly code file and then accessed via the HP-UX shared memory system functions. These functions are called from a C helper function. Use the labels under the examples to note each piece of code.

```
global  _com
set _com,0x1d000
```

File com.s

Continue on the next page.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/shm.h>
# include <stdio.h>

extern int com;
int shmid;

initshm()
{
 int *com_adr;

 if ( (shmid = shmget(ftok("f1.f"), 1024, 0666PC_CREAT)) == -1 )
  {
  fprintf(stderr, "problem with shmget\n");
  exit(1);
  }
 if ( (int)(com_adr = (int *)shmat(shmid, &com, 0)) == -1 )
  {
  fprintf(stderr, "problem with shmat\n");
  exit(1);
  }
 printf("shared memory (id=%d) located at 0x%x should be 0x%x\n",
  shmid, (int)com_adr, (int) &com);
}

pause()
{ sleep(1);
}
```

The C helper functions in file initshm.c

Continue on the next page.

```
  program F1
  integer i, k(13)
  common /com/ k
  call initshm
  do i = 1,13
k(i) = k(i) + 2
write (*,*) "F1: ",k(i)
call pause
  end do
  end
```

The first FORTRAN Program in file f1.f.

```
  program F2
  integer i, k(13)
  common /com/k
  call initshm
  do i= 1,13
k(i) = k(i) - 3
write (*,*) "F2: ", k(i)
call pause
  end do
  end
```

The second FORTRAN Program in file f2.f.

Continue on the next page for compilation and so on.

The above files are compiled, assembled and executed with the following Makefile using the command **make**:

```
all: f1 f2
        f1&
        f2&
f1: f1.o initshm.o com.o
    f77 -v -o f1 f1.o initshm.o com.o
f2: f2.o initshm.o
    f77 -v -o f2 f2.o initshm.o com.o
f1.o: f1.f
    f77 -v -c f1.f
f2.o: f2.f
    f77 -v -c f2.f
com.o: com.s
    as com.s
initshm.o: initshm.c
    cc -v -c initshm.c
```

File Makefile

The assembly file **com.s** reserves the shared memory space. The C function **initshm()** associates this space with the two processes created when files **f1** and **f2** are executed (in background mode).

Index

MANUAL COMMENT CARD

HP-UX Portability Guide

HP Part Number 97033-90046 7/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

Particular pages with errors? ______________________

____

Comments: ______________________

____

____

____

____

____

____

Name: ______________________

Job Title: ______________________

Company: ______________________

Address: ______________________

____

☐ Check here if you wish a reply.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
97033-90046

Microfiche No. 97033-99046
Printed in U.S.A. 7/87

97033-90614
For Internal Use Only