HP-UX 11i
Technical
Whitepaper

Node and Host Name Sizes on HP-UX

Using the Expanded Capabilities of HP-UX

Revision 2.0

ABSTRACT

HP-UX 11i v2 and v3, as well as later versions, provides expanded system node and host name interfaces. This capability allows longer node and host names than were previously supported. HP has provided this enhancement to support the system naming conventions used by some customers.

This paper is intended for system administrators and program developers. For the administrator it describes how to activate the capability, and describes the issues that must be addressed before increasing the size of the system node and host names.

For the developer it describes how to determine whether programs have dependencies on node or host name sizes, and how to enhance them to accommodate expanded node and host names.

Table of Contents

Introduction

System node and host names on HP-UX have default length limits of 8 and 64 bytes, respectively. The system administrator can configure the system to expand both these limits to 255 bytes.

Some customers who have thousands of systems use a naming convention to clearly identify the physical location and purpose of each machine. For instance, the convention might encode: the geographical location, the computer room name, the purpose (such as application), and an index. Allocating 2 bytes for each field fills the default HP-UX limit of 8 bytes. But two bytes might be too restrictive for some customers (such as when there are more than 100 systems for a particular purpose in one computer room, the index must be 3 bytes).

Programs that use the system functions to obtain (or initialize) the host and node names, and that are built prior to availability of the expanded interfaces, very likely require that names not exceed the non-expanded limits for proper operation. Therefore, the HP-UX default configuration ensures that the default runtime environment is compatible for existing application binaries.

This paper describes how to activate the expanded-name capability, the risks to applications, limitations regarding HP product support of long names, how to determine whether software code has dependencies on the default name size limits, and how to enhance programs to accommodate expanded name sizes.

- The section “Activating and Setting Expanded Node and Host Names” describes, for system administrators, how to enable and use the expanded name capability. Appendix A “Rules in Constructing Host and Node Names” contains some related information.

- The section “HP Product Limitations” describes the limitations some HP products have with respect to the size of node and host names.

- The section “Programmer Notes” describes briefly how software developers can enhance their applications to accommodate expanded node and host name sizes. Appendix C “Source Code Issues with Expanded Node and Host Names” and Appendix D “FLV Concepts and Usage” provide more information for programmers.

- A glossary follows the appendixes.

Caution: Assignment of node names in excess of 8 bytes or host names in excess of 64 bytes can result in incorrect operation or failure of some applications. It is important to read and understand all the information in this document before you try to use expanded name sizes.

Publication History

Revision 1.0 May 2005, HP-UX 11i v2 (B.11.23.01) This version was included in the NodeHostNameXpnd optional product bundle.

Revision 2.0 January 2007, HP-UX 11i v3 (B.11.31.01). The primary differences for 11i v3 are: 1) node/host name expansion is part of core OS product, no optional product is required; 2) the non-expanded uname(2) system function will return EOVERFLOW if the node name cannot fit in the non-expanded field (is more than 9 bytes including null terminator); 3) added list of minimum versions of optional products (was in product note for NodeHostNameXpnd bundle); 4) application core file format is no longer selectable.

Activating and Setting Expanded Node and Host Names

This section describes how to activate the capability to set longer node and host names, and how to actually set the longer names.

Note the following important considerations:

- You must verify that the versions of all applications that use node or host names are validated for expanded node and host name sizes before setting the names to longer values.
- When longer node or host names are assigned, it is recommended that proper operation of all applications be validated in a test environment before use in a production environment.
- Some HP products are limited in their support of long node or host names (see “HP Product Limitations”).
- Third-party software products might be limited in their support of long node names or host names. Consult the associated documentation for products to ensure that they do support expanded names.
- Locally developed programs (for example, from an in-house software development organization) might offer limited support of long node or host names.
- The behavior of the nonexpanded version of the `uname(2)` system function when the system node name is set to a value greater than 9 bytes (including null terminator) is configurable. It can be configured to return the EOVERFLOW error, which alerts the program that the data cannot be returned (this is the default). Any program using the nonexpanded version will likely abort, unless it is ignoring errors.
- The system can be configured to return truncated data rather than an error from the non-expanded `uname(2)` system function. Such truncated data might allow the program to continue with no visible abnormal behaviors. On the other hand the program might provide ambiguous outputs or behave in some indeterminate manner. Programs updated to use the expanded version do not exhibit these problems.
- Any program that uses the `gethostname(2)` system function and that is not enhanced to accommodate the expanded node and host name sizes, receives a truncated host name from this interface. The value is truncated to the size of the program buffer, which is typically 64 bytes in non-enhanced programs. If the program is written to detect truncation, it might abort on systems where the host name is set to a value longer than 64 bytes. For other programs, undetected truncation might result in ambiguous output or indeterminate program behavior.
- Any program that uses the utsname structure definition and that is not enhanced to accommodate the expanded structure version might exhibit ambiguous output or other indeterminate behavior.
- Any program that internally uses the symbolic constants `SYS_NMLN`, `SNLEN`, `UTSLEN`, or `MAXHOSTNAMELEN` can have issues similar to those described in the preceding paragraphs.
- For complete information about application issues, see the programmer-oriented sections and appendixes of this document.

Activating Expanded Node and Host Names

The default operating system configuration does not allow you to set a node name to greater than 8 bytes, or a host name to greater than 64 bytes. A dynamic kernel tunable parameter, **`expanded_node_host_names`**, must be turned on to allow larger names to be set. It is documented in the manpage `expanded_node_host_names(5).` To turn on this parameter, use the following command:

```
kctune expanded_node_host_names=1
```

To turn off this parameter, use the same command, but set the value to `0`. For more information, see `kctune(1m)`. Note that changing this parameter does not affect any current node and host name settings, regardless of their length. The node and host names should be changed to 8 or 64 bytes (or less) before turning off the parameter.

Setting Nonexpanded uname(2) Overflow Reporting

The default operating system configuration causes the nonexpanded version of the `uname(2)` system function to return the EOVERFLOW error if the system node name is set to a value of more than 9 bytes (including null terminator). A tunable parameter, **`uname_eoverflow`**, is provided to control whether or not this error is reported.[1] It is documented in the manpage `uname_eoverflow(5)`.To change the system behavior to silently truncate the node name without an error, use the following command:

```
kctune uname_eoverflow=0
```

Caution: The lack of error reporting means that applications are not alerted to inaccuracy of the requested data (that is, truncation). Therefore, the application might exhibit other indeterminate failures.

Setting a Node or Host Name

Once the `expanded_node_host_names` tunable is turned on, you can set a longer node or host name using the same utilities that are used to set names of any size.

To set the node and host name, and to make the settings persist after a reboot, use this command:

```
/sbin/set_parms hostname
```

Note that you must type the string `hostname,` not the desired host name. This command prompts for the desired host name. It then edits the system configuration scripts to ensure that the names persist after a reboot.[2]

In some cases you might want to make the node and host name settings temporary (that is, the settings do not persist after reboot). To do so, use the following commands for the node and host names, respectively:

```
setuname –t –n <nodename>
hostname <hostname>
```

Note that it is not possible to set long names during system cold installation. This is because the `expanded_node_host_names` tunable is off by default. Temporary node and host names should be assigned until the longer names can be set. (The network administrator should be able to assign a short alias for the long name.)

[1] This tunable is first available in HP-UX 11i v3. HP-UX 11i v2 provided no capability to report the error.

[2] Note that `set_parms` does not accept individual labels in the name in excess of 63 characters. For more information see Appendix A "Rules in Constructing Node and Host Names."

HP Product Limitations

This section describes the limitations of HP software products in support of node names longer than 8 bytes or host names longer than 64 bytes. Some of these limitations might be removed in product or OS updates made available after the publication of this document. Providers of third-party and proprietary software can also have some limitations.

Note that this list might not be complete. Any other important limitations encountered by customers should be reported to HP.

C/C++ Compilers

The Function-Level Versioning (FLV) feature of the C/C++ compilers is required to support application program updates for expanded node and host names. The following table lists the minimum compiler versions that support FLV. Earlier versions do not support FLV and, therefore, do not support compilation of programs updated for expanded node and host names.

Optional Product Description	Bundle/Product Name	Minimum Version Needed for Node/Host Name Expansion
HP C/aC++ Developer's Bundle - PA C	B9007AA	B.11.23.08
HP C/aC++ Developer's Bundle - PA aC++	B9007AA	C.03.62
HP C/aC++ Developer's Bundle - IPF	B9007AA	A.06.00

Client Systems (Various Network Utilities)

Versions of HP-UX prior to 11i version 2 May 2005 Update do not support expanded host names. Therefore, various client utilities on prior HP-UX versions might not be able to access servers that have very long host names. For example, utilities such as `remsh(1)`might not be able to access servers with canonical names (names with domains added) longer than 63 bytes. Similarly, NFS client systems having earlier OS versions installed cannot access NFS servers having names longer than 31 bytes (excluding null terminator). Therefore, HP-UX NFS server systems should not be assigned longer names unless all HP-UX client systems that access that server have been updated.

Common Desktop Environment (CDE)

In configurations with a long host-name set, graphical user interface applications might not display the entire host name. The number of bytes in the host name that are actually displayed can vary with display and font size.

Expanding the window size in some applications (such as, `dtfile` – CDE file manager) shows the first component of the host name. The CDE login window cannot be resized.

Core Files

The application core file format on HP-UX 11i v3 has been updated so that it can accommodate the maximum node-name length.

This updated format was introduced as an option with HP-UX 11i v2 (and likely used only on systems with the optional NodeHostNameXpnd product bundle installed). Core files generated with the prior format contain only up to the first 8 bytes of the node name. Debuggers do not have the whole node name when accessing such core files.

File Names

File names in the default HP-UX file system configurations have a maximum length of 255 bytes. This maximum cannot accommodate names created by applications that include the host name with a short prefix or suffix, when the total length of the host name, prefix, and suffix exceeds 255. Host names generally do not get sufficiently long to make this a problem.

Note also that the maximum length of a full path name equals the value of the `PATH_MAX` parameter (usually 1024 bytes). In a small number of cases (CDE is one example), the host name is used in a directory name superior to a file name that also contains the host name. While this path does not exceed the `PATH_MAX` value, the value can be exceeded if the host name appears multiple times in the path of some application file.

Some file system configurations support file names only up to 14 bytes. These systems should not be assigned host or node names longer than 8 bytes so that the node or host name can be used in the formation of file names within some applications.

Ignite-UX

Ignite-UX supports node and host names up to 63 bytes (plus null terminator).

Internet Services

The `/etc/gated.conf(4)` and `/etc/dhcpv6tab(4)` configuration files (used by `gated(1m)` and `dhcpv6d`, respectively) support tokens up to 200 bytes (including null terminator). Setting a fully qualified host name for the system could run into this limitation. However, this is unlikely because fully qualified host names generally do not get that long. A future version of HP-UX might remove this limitation.

The mail ID size supported by `sendmail(1m)` has been increased from 255 to 511 bytes to accommodate maximum-size host names. However, mail being routed through systems not capable of handling these sizes (such as systems prior to HP-UX 11i v2 September 2004 Update) might result in delivery failures. This is very unlikely, because fully qualified host names generally do not get sufficiently large to cause mail IDs to exceed the original 255-byte limit.

The `sendmail` utility can create files that include the host name (such as `.forward.hostname+`). With longer host names, it is somewhat more likely that the path to these files can exceed the `PATH_MAX` value, thus causing failures. Because the `PATH_MAX` value is 1023, this should be a problem only when the directory path itself already exceeds 756 bytes. For more information, see "File Names".

The standard BOOTP protocol supports host names up to 64 bytes (including null terminator). The server truncates any name that is longer. Because DNS standards limit host name labels to 64 bytes (including null terminator), this is generally not an issue when the name is not qualified with the domain name. In cases where the label or qualified name exceeds this limit, a shorter host-name alias might be assigned and used with the BOOTP utilities.

The `rwho` protocol supports host names up to 32 bytes. Longer host names cannot be supported in `rwhod(1m)` and `rwho(1)` and should not be assigned on systems running the `rwhod` system daemon.

Some utilities, such as `remsh(1)`, internally append the domain name to the host name, if it is not already specified, to form a full canonical name. This full name cannot exceed 255 bytes (excluding null terminator). This is not a problem as long as host names conform to the Internet rules summarized in Appendix A "Rules in Constructing Host and Node Names".

Java

The following table lists the minimum Java™ versions required for support of expanded node and host names. Earlier versions do not have full support.

Optional Product Description	**Bundle/Product Name**	**Minimum Version Needed for Node/Host Name Expansion**
Java 1.4.2 JDK	T1456AA	1.4.2.07
Java 1.4.2 JRE	T1457AA	1.4.2.07
Java 1.3.1 JDK	B9788AA	1.3.1.15
Java 1.3.1 JRE	B9789AA	1.3.1.15

Languages Other Than C/C++

Only the HP C and C++ compilers support the FLV mechanism used for `uname()` as described in this document.. Other language compilers and linkers do not support that mechanism. Programs written in other languages must use explicit source code sequences to access the versioned `uname()` function.

Serviceguard

The Serviceguard product supports host names up to 39-bytes (not including null terminator).

Shared LVM

In a Shared LVM SGeRAC environment the `vgdisplay` command displays the host names of activated/shared volumes. In this environment it supports up to 39 bytes (not including null terminator).

UUCP

UNIX-to-UNIX Copy Protocol (UUCP) on HP-UX supports node names only up to 8 bytes. The intersystem protocols cannot support longer node names. If the system node name is set to more than 8 bytes, UUCP might fail to function properly.

VxVM

The VxVM volume manager product supports node names up to 8 bytes.

Programmer Notes: Overview for Updating Programs

Most application code is unaffected by the node and host name expansion. However, some applications need to be recompiled with options that select name expansion. In a subset of those cases, some code modification might be necessary.

This section describes how software developers can determine whether a product might need modification and, if so, how to enhance the product to accommodate expanded node and host names. Enhanced programs also operate correctly in systems where the names are not long. Programs that have no dependencies on node or host name size, or that are enhanced for the expanded sizes, are said to be expanded node and host name clean.

Developers should read this section in its entirety, along with the related appendixes, in order to understand all the issues. Developers should also become familiar with the administrative issues described in "HP Product Limitations".

Is The Program Already Clean?

Programs in which the source code does not use any of the following symbols are already clean and need no further consideration:

For node names: `sys/utsname.h, utsname, utsname.nodename, uname(),SYS_NMLN,SNLEN,UTSLEN`

For host names: `gethostname(), sethostname(), MAXHOSTNAMELEN`

Note that some programs use literal constants when one of the preceding symbolic constants (those with full capitalization) should have been used instead. These programs might pass a symbol search but still do not support the expanded size for node or host names.

The Basics

Most programs that are not already clean simply need to be recompiled to use expanded versions of the interfaces. This entails adding the compiler option `-D_HPUX_API_LEVEL=20040821` to the makefile.[3] This does the following:

- Selects expanded values for `MAXHOSTNAMELEN, SYS_NMLN, SNLEN, and UTSLEN`.
- Defines an expanded `utsname` structure.
- Causes source references to `uname()` to link to the expanded version of the function in `libc`.

All elements (source files) of a program that reference any of the preceding symbols must be recompiled to ensure they all have compatible data layouts. If a function in one compilation unit passes a buffer of `MAXHOSTNAMELEN` bytes to a function in another compilation unit, it is important to ensure both functions are using the same size. The same concern applies to the other symbols as well.

[3] Note that this can also be defined in the source file itself, with `#define _HPUX_API_LEVEL 20040821`. However, it must be the first line of the source (it must precede any `#include` directives), and it must appear in all source files that comprise the program. Specifing the compiler option in the makefile is generally less error-prone. Note also that future versions of HP-UX may define API levels with identifiers greater than `20040821`. These future versions are cumulative and will include all the features of API level `20040821`.

Note that it does not have to be a system with the tunable `expanded_node_host_names` turned on in order to build with this option or to run the resulting binary. However, the resulting binary might be able to execute correctly only if HP-UX 11i v2 September 2004 Update or higher is installed, as that update contains the expanded programming interfaces. Furthermore, if the program does not use node-name related symbols (that is, if it uses only host-name related symbols), it can also execute correctly on the original HP-UX 11i v2 2003 version. However, if you want to execute on the original HP-UX 11i v2, it is best to use the technique described in "Building for Execution on Multiple Releases.".

Note also that enhancing programs to use the expanded `utsname` structure and `uname(2)` system function prepares them to handle future value expansion for any other fields in that structure.[4]

Behavior of Existing Interfaces

The interfaces used to obtain the node and host names are `uname()` and `gethostname()`, respectively.

In the case of `uname()`, an application that has not been updated to use the expanded version of the function will be using a version of the associated `utsname` structure which has a `nodename` field of 9 bytes (for 8 byte node name and a terminating null byte). In an environment where the system's node name has been set to more than 8 bytes, the program either gets an EOVERFLOW error or receives only the first 8 bytes of that name. The reporting of the EOVERFLOW error is activated or suppressed by the `uname_eoverflow` system tunable parameter.

Caution: On HP-UX 11i v2 an error status is not returned to indicate that the truncation has occurred. The lack of an error return on truncation occurs because many programs use the `uname()` function for data other than the node name, or can easily tolerate using the truncated value. This changed in 11i v3, so that applications are notified of the incomplete data as soon as possible.

In the case of `gethostname()`, the returned string length is limited to the size provided as the second parameter. Many programs allocate buffers using the symbolic constant `MAXHOSTNAMELEN` (although it is important to note that some use smaller constants). For the default compilation environment, its value is 64. Adding `-D_HPUX_API_LEVEL=20040821` to the compilation environment changes its value to 256, thus accommodating larger host names. For programs that are not recompiled and run on a system with a host name longer than 64 bytes, only the first 64 bytes will be returned. No error status is provided because the standard specifications state that the returned name shall be truncated if the supplied buffer is of insufficient length. On HP-UX, the lack of a terminating null byte in the buffer indicates that the name might not be complete.

Source Code Issues and Fixes

In the vast majority of cases, compiling as described in the preceding section makes the program long node and host name clean. However, problem areas might exist in some programs that need source code corrections. For the most part, these problems stem from code that does not use the interfaces as specified. That is, latent bugs might be exposed by the expanded node and host name interfaces.

[4] No field value other than `utsname.nodename` is expanded for the initial release of HP-UX11i V3. While there are no plans for the near future, it is possible that other values in that structure will be expanded in an update or succeeding release.

All programmers are advised to read Appendix C, "Source Code Issues with Expanded Node and Host Names," and inspect their programs for actual usage of the names, to ensure the program operates correctly when recompiled.

Building for Execution on Multiple Releases

This section describes how to build a program to handle expanded node or host names to be executed on systems regardless of release or update level.

Building for Any HP-UX 11i v2 or Higher System

If a program is compiled with `-D_HPUX_API_LEVEL=20040821` and the node-name related interfaces are used, the resulting binary might not be portable to HP-UX 11i v2 systems that do not have the 11i v2 September 2004 Update or later installed. However, programmers can take additional measures to ensure the binary program runs on the original, nonupdated, 11i v2 system:

- Do not use the compiler option `-D_HPUX_API_LEVEL=20040821.`
- Use `gethostname(2)` in place of `utsname.nodename` (you might need to scan up to the first dot (".") if a fully qualified domain name is not desired).
- If the `uname()` function is used for data other than the node name, the program should be prepared to deal gracefully with the EOVERFLOW error status which indicates that the whole node name did not fit into the `utsname.nodename` field and has been truncated.
- Do not use the `MAXHOSTNAMELEN` symbolic constant. Instead, define a private symbolic constant to set the size of buffers and the size parameter passed to `gethostname().` The value must be at least 256.
- Build on any HP-UX 11i v2 system, regardless of update level.

The resulting application binary should run correctly on any HP-UX 11i v2 system and all subsequent versions. It is also expanded node and host name clean, which means it accommodates any current and future version of HP-UX that supports expanded node and host names and for which the system administrator has set long names.

Do not use this approach for programs that re-export, via an API, the `utsname` structure, or any of the symbolic constants: `MAXHOSTNAMELEN, UTSLEN, SYS_NMLN, SNLEN` (or equivalents).

Caution: For programs that use `utsname` structure fields other than `nodename`, do not ignore the EOVERFLOW error. A future HP-UX update or release might expand the possible values of other fields. The program should determine whether or not the values obtained are complete. Furthermore, in a future version of HP-UX, generation of the EOVERFLOW error might also inhibit the return of any data, whether or not it is truncated. For additional information about this check the release notes that accompany updates and releases.

Building for Any HP-UX Version

Many programmers prefer to create a single application binary to be deployed on any HP-UX OS release. For instance, a PA-RISC binary can be built on HP-UX 11.00, and be deployed on that release and on HP-UX 11i v1, v2, and future versions. An Intel® Itanium®-based binary can be built on HP-UX 11i v1.6, and be deployed on that version, HP-UX 11i v2, and on future versions.

However, since the expanded version of the `uname(2)` function and associated structure does not exist on versions prior to HP-UX 11i v2 September 2004 Update, it is not possible to build a binary that uses expanded `uname(2)` on those versions.

The solution is to employ the approach described in the previous section, and to compile the program on the earliest HP-UX version on which it is to be run.

Again, do not use this approach for programs that re-export, via an API, the `utsname` structure, or any of the symbolic constants: `MAXHOSTNAMELEN, UTSLEN, SYS_NMLN, SNLEN` (or equivalents).

Dynamic Buffer Sizing

Programs should avoid using the `MAXHOSTNAMELEN` parameter. Instead, they should query the system for the maximum possible length by using `sysconf(_SC_HOST_NAME_MAX).` The result, after adding 1 to allow for a null terminator, should be used to allocate a buffer of that size to hold the host name.

Because this system configuration variable is not available on all versions of HP-UX, you might want to use conditional compilation in the following manner:

```
      hostbufsz = 0;
#ifdef _SC_HOST_NAME_MAX
      hostbufsz = sysconf(_SC_HOST_NAME_MAX) + 1;
#endif
      if (hostbufsz <= 0) hostbufsz = 256;
      hostbuf = malloc(hostbufsz);
      retval = gethostname(hostbuf, hostbufsz);
```

The code fragment defaults to a host name buffer size of 256 when the `sysconf()` parameter is not available.

Re-export of Node Name and Host Name Interfaces

Some software developers supply libraries of application interfaces for their customers. These “exported” interfaces are accessible to customer applications. These interfaces might be functions or accessible data structures.

The specification of these interfaces can include direct dependencies on the HP-UX node or host name interfaces. If so, they are said to “re-export” those interfaces. Examples of re-exported interfaces include:

- A structure, `struct abdx`, is passed into or out of some library function. A field in `struct abdx` is `struct utsname abdx_uts`. Thus, the interface re-exports the `utsname` structure.
- A library function, `muddle()`, accepts a pointer to a buffer into which it places a string. The specification states that the buffer should be `UTSLEN` bytes long. This re-exports the `UTSLEN` symbol.

If the library functions are simply recompiled with `-D_HPUX_API_LEVEL=20040821,` they can accommodate long host and node names. They can also allow the applications that use them to recompile to accommodate the long names.

However, the rebuilt interfaces will not be binary compatible with existing client binary programs. Existing application binaries do not have a compatible layout for `struct abdx`. Calling `muddle()` could result in an application buffer overrun and undetermined behavior.

The library provider has a choice, either the client programs must also be recompiled (remember that all elements of the program should be compiled in the same manner); or the library provider can supply two versions of the affected interfaces (one that is nonexpanded and compatible, and an expanded one that accommodates long names).

The FLV mechanism enables the library developer to provide multiple versions of interfaces. For more information see Appendix D, “FLV Concepts and Usage”.

Summary

HP has provided expanded node and host names in response to the needs expressed by customers. This feature increases the maximum node and host name lengths from 8 bytes and 64 bytes, respectively, to 255 (plus terminating null) for both.

Because of the interface changes this involves, an interface version (or level) has been introduced that includes expansions for the node name and host name-related APIs. The associated system commands and utilities have been updated to use the expanded APIs themselves.

However, longer names must be used with extreme caution. Programs that are not rebuilt to accommodate the expanded name interfaces will truncate the names or receive an error. This can cause programs to exhibit ambiguous output or incorrect behavior (including unexpected and undetermined aborts). In order to avoid these problems, the default system and product configuration settings limit names to the original, compatible, maximums. The system administrator must explicitly enable the expanded name capability through a kernel tunable.

This paper has described how to activate the capability, limitations of specific HP products, how software developers can determine whether their products have dependencies on node or host name sizes, and how to enhance the programs to accommodate long names

APPENDIX A: Rules for Constructing Host and Node Names

This appendix describes some rules, established by Internet standards bodies, for the formation of host names.

The Domain Name Service (DNS), as documented in RFC 1034, limits a fully qualified host name to a total of 255 octets (bytes). A name is composed of labels separated by dots (.) where each label is limited to 63 octets. A label can be composed of letters, digits, and hyphens (-). Letters are treated in a case-insensitive manner. A label cannot start with the hyphen. It is unclear whether a label can start with a digit, so do so only with caution.

System administrators usually set the host name on the system to just the first label (without dots). This is generally more convenient for users and applications. This is considered a “relative” name (that is, relative to its Internet domain).

HP strongly recommends that you set the system node name to the same value as the first label set in the system host name. This is because many programs use the node name as the Internet host name. For example, if the host name is set to sys.x.y.z, the node name should be set to sys. In this way node names will not be more than 63 bytes for systems with host names that conform to the standards.

The following table provides additional examples:

Host Name	**Node Name**
jims-test-system.dev.xyzco.com	jims-test-system
payrollsystem1	payrollsystem1

APPENDIX B: HP-UX Node and Host Name Interfaces

This appendix provides additional information about how the OS interfaces that directly manipulate the node and host name are affected by the support of expanded names. In some cases, the changes were effective in 11i v2 September 2004 Update. In all cases, the default configuration environment will be compatible with that of HP-UX releases and updates prior to the 11i v2 September 2004 Update (8-byte node names and 64-byte host names).

Functional Program Interfaces

`sethostname(2)`

This function accepts up to 255 bytes (plus null terminator) for the host name if the `expanded_node_host_names` tunable is 1 (on). Otherwise, it accepts up to 64 bytes (plus null terminator).

`gethostname(2)`

This function returns up to 255 bytes (plus null terminator) of the host name previously set via `sethostname(2)` on the 11i v2 September 2004 Update and succeeding updates/versions. Otherwise, it returns up to 64 bytes (plus null terminator) on prior versions of 11i.

`setuname(2)`

This function accepts up to 255 bytes (plus null terminator) for the node name if the `expanded_node_host_names` tunable is 1 (on). Otherwise accepts up to 8 bytes (plus null terminator). Other capabilities are unchanged.

`uname(2)`

This function returns an `utsname` structure. Two versions of this function and corresponding structure are available. The nonexpanded version is consistent and compatible with that of previous HP-UX releases. All fields are 9 bytes except for one which is 15 bytes. The expanded version, with 257-byte fields, is available on the 11i v2 September 2004 Update and succeeding updates/versions. It is used if the program is compiled with `_HPUX_API_LEVEL` set to `20040821`.

For the nonexpanded version, if the system node name is set to more than 9 bytes (including null terminator) it returns an EOVERFLOW error if the `uname_eoverflow` system tunable is set to 1 (the default). Otherwise, the error is not reported and the value is truncated to 8 bytes (plus null terminator). In a future release, no data will be returned if EOVERFLOW is reported.

`sysconf(_SC_HOST_NAME_MAX)`

This function returns the value of the system variable `HOST_NAME_MAX`. This is the maximum size of a host name that the OS configuration can support. Programs can use this function to provide a size for buffers to hold the host name. It returns 255 (not including null terminator) on 11i v3 and on 11i v2 if the optional NodeHostNameXpnd product bundle has been installed; otherwise, it returns 63. This `sysconf(2)` variable is not available prior to 11i v2 September 2004 Update. Programs receive an EINVAL error if used on earlier versions.

Programmatic Constant Interfaces

SYS_NMLN, UTSLEN

These are symbolic constants used to determine the size of fields in the utsname structure. Their values are 257 if the program is compiled with –D_HPUX_API_LEVEL=20040821. Otherwise, their values are 9.

SNLEN

This is a symbolic constant used to determine the size of the idnumber field in the utsname structure. Its value is 257 if the program is compiled with –D_HPUX_API_LEVEL=20040821. Otherwise, its value is 15.

MAXHOSTNAMELEN

This is a symbolic constant used to determine the maximum size of a host name. Its value is 256 if the program is compiled with –D_HPUX_API_LEVEL=20040821. Otherwise, its value is 64.

The utsname Structure Interface

The utsname structure is defined as:

```
struct utsname {
     char    sysname[SYS_NMLN];
     char    nodename[SYS_NMLN];
     char    release[SYS_NMLN];
     char    version[SYS_NMLN];
     char    machine[SYS_NMLN];
     char    idnumber[SNLEN];
};
```

When the SYS_NMLN and SNLEN values change for the expanded version (that is, when the program is compiled with –D_HPUX_API_LEVEL=20040821), this structure exhibits a different size and layout. (In addition, it includes fields added for future expansion.)

For HP-UX 11i v2 and v3, only the nodename value is permitted to exceed the length allowed by prior versions of HP-UX. The other field values cannot exceed the prior limits.[5] However, future needs are not always predictable, so HP might expand other values so that they do not fit in the corresponding fields of the nonexpanded version of the structure. This will cause the nonexpanded version of uname(2) to return an EOVERFLOW error.

[5] Why have other values not been expanded? There is little call for a longer sysname field (it is typically "HP-UX"). The release, version, and machine fields cannot be controlled by the customer. HP does not want to force longer strings on applications without customer control. The release field is parsed by many programs and scripts which could not tolerate a deviation from the normal format (such as B.11.11). The version is a single byte that encodes the license level. The machine field is also parsed by programs and scripts that cannot tolerate a deviation from the normal format (such as 9000/889). [Alternate interfaces exist for more complete model information (see model(1), getconf(1) w/MACHINE_MODEL, confstr(3) w/_CS_MACHINE_MODEL). Finally, the idnumber field must be parsable by into a 32-bit unsigned integer value so it cannot exceed 10 bytes (representing 10 decimal digits).

Command Interfaces

`uname(1)`

The `uname` command displays up to 256 bytes of the node name, when invoked with the `–n` or the `–a` option (effective in HP-UX 11i v2 September 2004 Update). It allows setting of the node name with the `–S` option, up to 256 bytes (plus null) when the `expanded_node_host_names` tunable is 1 (on), or up to 8 bytes when the tunable is 0 (off).

`setuname(1m)`

The `setuname` command allows setting of the node name with the `–n` option, up to 256 bytes (plus null terminator) when the `expanded_node_host_names` tunable is 1 (on), or up to 8 bytes when the tunable is 0 (off).

`hostname(1)`

The `hostname` command displays up to 255 bytes of the host name when invoked with no options (effective in the HP-UX 11i v2 September 2004 Update). It allows setting of the host name, up to 255 bytes (plus null terminator) when the `expanded_node_host_names` tunable is 1 (on), or up to 8 bytes when the tunable is 0 (off).

APPENDIX C: Source Code Issues with Expanded Node and Host Names

This appendix describes potential issues that might need to be addressed in existing programs.

General Program Issues

This section describes problems that application programs might exhibit that require attention from developers. In most cases the solution is to recompile to the API level that includes the expanded interfaces (that is, `_HPUX_API_LEVEL=20040821`). In some cases, the code is not written strictly to the specification and must be changed before recompilation (such as to use the proper symbolic constants).

This section is not an exhaustive study of all possible problems. It describes the basic classes of issues, which software developers can use as models for the situations peculiar to their own products.

Truncated Node or Host Name Fields

With the `expanded_node_host_names` tunable turned on, the system administrator can set node or host names up to 255 bytes long. The expanded version of the OS interfaces provides these names, in their entirety, to their callers.

Any software that still uses the nonexpanded versions of the node name interfaces (functions, constants, or structures) either gets an EOVERFLOW error or observes that the values set by the administrator are truncated to the first 8 bytes, depending on the setting of the `uname_eoverflow` tunable.

Software that still uses the nonexpanded versions of the host name interfaces (that is, `MAXHOSTNAMELEN`) observes that the host name value set by the administrator is truncated to the first 64 bytes.

The truncated names can be confusing if displayed to the user or printed in reports. Log files can be ambiguous, especially if names on multiple systems have the first 8 or 64 bytes of the names the same. For this reason, it is advantageous to arrange the naming convention to make the first 8 bytes as unique as practical.

Because the name is truncated, the program's notion of the local host's name is incorrect. Passing the truncated name outside of the system (such as passed in a network protocol, or via another communication media) can cause later attempts by external systems to establish a network connection to fail (or, perhaps worse, can result in a connection to the wrong host).

Parsing Output of uname(1) or hostname(1) Commands

Scripts or programs that parse or accept the output of the `uname(1)` or `hostname(1)` commands might make assumptions about the width of the fields. These assumptions need to be removed.

Though unlikely, there might also be assumptions that, in the output of `uname -a`, the node name field starts in the seventh column. This is already an invalid assumption, because the first field is not required to be the string “HP-UX” (although it usually is). Such an assumption needs to be removed.

Arrays Dimensioned by UTSLEN, SYS_NMLN, SNLEN, or MAXHOSTNAMELEN

A program might use one of the symbolic constant interfaces (`UTSLEN, SYS_NMLN, SNLEN, or MAXHOSTNAMELEN)` to declare the size of array buffers to hold the node or host name. If not rebuilt with the compiler option `–D_HPUX_API_LEVEL=20040821`, the program's buffer is not large enough to hold the entire node or host name.

In most cases, when recompiling to use the expanded version, these symbol references adapt cleanly to the expanded values. However, developers need to validate this for their code.

Because `UTSLEN, SYS_NMLN, and SNLEN` are used in conjunction with `uname(2)`, and because the program must be rebuilt to use the expanded `uname(2)` function version, the recompile is usually the best approach. However, with `MAXHOSTNAMELEN`, it is possible to simply not use it, and have the program simply define its own buffer sizes to pass along to `gethostname(2)`. See "Building for Execution on Multiple Releases". Rebuilding the program is still required, but there is more flexibility in compilation environment and in deployment.

Arrays Dimensioned by Constant 8, 9, 15, or 64

Some programs do not use the symbolic constants but instead use hard-coded integer constants to declare arrays that ultimately hold values returned by `uname(2)` or `gethostname(2)`. Strictly speaking, this is a violation of the specification.

These programs must be modified to use the symbolic constants before recompiling to use the expanded interface version. (Better yet, use `sysconf(_SC_HOST_NAME_MAX)` to obtain the maximum size, and dynamically allocate buffers to hold the host name. For a sample code fragment see "Dynamic Buffer Sizing".)

Embedded utsname Structure or Node/Host Name Constants

Some programs use the `utsname` structure or one of the node name or host name constants (`SYS_NMLN, SNLEN, UTSLEN, or MAXHOSTNAMELEN`) in the definition of their own data structures. That is, the `utsname` structure might be a substructure of a program data structure. Alternatively, one of the listed constants (or their literal equivalents) might be used to declare an array within a data structure. For the purpose of this discussion, we say one interface is "embedded" within another.

The concern about node and host name interfaces embedded within a program data structure depends on the program's use of that data structure.

Embedded in Internal-Only Data Structures

The node/host name structure/constant interfaces may be embedded inside program private structures. Generally, such programs can be updated for expanded names simply by recompiling with the option `–D_HPUX_API_LEVEL=20040821.` The alternative approach for `MAXHOSTNAMELEN` described in "Building for Execution on Multiple Releases" is also acceptable.

Embedded in Storage Formats

Sometimes the node/host name structure/constant interfaces are embedded in storage formats (such as files, tapes, and so on). Recompiling the program to use the expanded version has two problems. First, the program can no longer read files created with the prior version of the program. To remedy this, the program needs to recognize that it has encountered a nonexpanded format, and it must be able to use the nonexpanded structure layout to read it. Updating the the stored format to the expanded version is optional.

Second, if other programs are expected to read the expanded stored format, they must be updated to use the expanded version. If the other programs are user programs, this can be compatibility issue if older user programs need to be supported. One solution might be to always write two versions of the stored data. The nonexpanded version can be used by some programs, the expanded version can be used by enhanced user programs.

Some storage formats have flexible formats where the sizes of the data are described by metadata stored with or alongside the actual data. It might be possible to add expanded “sections” which are generally ignored except by programs that can recognize them. If so, the data that depends on the node/host name structures or constants can be added to the data in an expanded section, rather than replacing existing sections.

Embedded in IPC Communication Protocols

Node/host name structure/constants might be embedded in communication protocols. This is similar to having them embedded in storage formats, except there is no persistent, nonexpanded, stored data. In most cases, protocol version levels are negotiated between the parties. This leads to a natural way for the programs to adapt to one version or another.

In many cases, simply recompiling all the programs that use the protocol is the solution.

Embedded in Shared Memory Structures

The node/host name structure/constants might be embedded in structures that are placed in shared memory. Again, the situation is similar to that of having them embedded in storage formats.

When accessed only by multiple instantiations of a single program, updating that program to the expanded version (that is, recompiling with the expanded version selected) is sufficient. This is usually the case.

When accessed by different programs, the differing data layout can cause problems. Therefore, all the programs that access the shared memory should be updated together, if possible. Since programs evolve, changes to the shared memory format are not uncommon. This is not an unusual situation.

Depending on the data structures used, supporting both the nonexpanded and expanded formats in shared memory might be possible. All writers must be updated to deal with both formats. However, readers can use either format.

If the shared memory is accessed by others (such as, by customer applications), developers must devise a reasonable migration plan. However, this situation is highly unusual.

Embedded in Exported Interfaces (Re-exported)

Through a library, an application might export a function (or data structure) interface that includes in its definition arrays or structures containing a node/host name structure/constant. That interface is said to re-export the node/host name interfaces. The developer has some choices:

- Rebuild all clients of the application library to use the expanded HP-UX node/host name interfaces. This satisfies the rule that all programs comprising an application be built in the same manner. The library and all clients are expanded node/host name clean.
- Do nothing. The library and clients support only the nonexpanded node and/or host name interfaces. This prevents the library and its client programs from fully supporting systems on which the administrator has set a long node and/or host name.
- Make some modest code changes so the library supports two versions of the interface. For data structure interfaces, clients that need to also support long node/host names must be changed to access the expanded version of the data. For function interfaces, two functions can be defined. One supports the nonexpanded node and host names, the other supports the expanded names. This can be accomplished by introducing a second function with a different name. Alternatively, the two functions can have the same name by using the FLV mechanism described in Appendix D, "FLV Concepts and Usage".

Buffer Overflow and Other Subtle Issues

The system interfaces provided for applications to obtain the node and host names do not overflow buffers (unless parameters to those interfaces are incorrectly provided). However, long node and host names can cause buffer overflows in applications.

Consider a program module to which is passed an internal data structure containing a pointer to a node or host name. Using the `strcpy()` function, this module attempts to make a copy of the name into its local buffer created using some constant for the size. If that constant is less than what is necessary to contain the name, the buffer will overflow. (Using `strncpy()` would have been better to avoid overflow, but the example assumes some existing code.)

This module, and perhaps the compilation unit in which it resides, shows no obvious reference to `gethostname()`, `utsname.h`, `uname()`, the `utsname` structure or to any of the symbolic constants `MAXHOSTNAMELEN`, `SYS_NMLN`, `SNLEN`, or `UTSLEN`. So, the module is easy to miss during source code investigations.

This illustrates an important point. Investigation of node and host name dependencies in source code cannot stop at just the source files that have references to the operating system symbols in the preceding paragraphs. It is important to follow the data path of any node or host name string defined by the program. A full search must also be made of all references to these program-specific data field names.

Node or Host Names as Program Input

A node or host name might be supplied to an application as input via command-line options or data files. If the names are longer than the program was built to handle, the names might be truncated, cause run-time error messages, or cause buffer overflows (such as, when `strcpy()` or related string-manipulation functions are used without regard to the accepting buffer size).[6]

This is similar to the issue described in the previous subsection. However, an additional problem is that the entire application source code might not exhibit references to any of the node and/or host name related symbols. Therefore, it might not be obvious that the names are used anywhere in the application. It is important to investigate the operational specification of an application as well as scan its source for the telltale symbol references.

[6] Note that buffer overflows can create a security problem (especially in setuid programs).

Scripts

Scripts that use the `uname(1)` or `hostname(1)` command typically do not have problems with long node or host names. Yet, problems can occur.

The script might parse the output of the `uname(1)` command to obtain a desired field. This is very unlikely because the `uname(1)` command provides options to obtain the individual argument, making parsing unnecessary. Furthermore, any parsing scheme must already deal with the variability of individual field widths. Given this, problems should not exist with parsing of the command output, except in the unlikely case that the node name is explicitly expected to not be more than 8 bytes.

A script might attempt to store a node or host name in a variable that is defined to hold only 8 or 64 bytes, respectively. This is also unlikely.

It is safest to scan product scripts for references to the node and host name-related commands. However, it is very unusual to find any issues that need to be addressed in support of long names.

Tools to Help

The following script can help to scan application source code for dependencies on node and/or host names or on related symbols:

```
#!/bin/ksh
#
# Script to search source files for dependencies on node/host
# name size.
# Accepts a list of text file names for the arguments.  (Does not
# support embedded spaces or other special characters in the
# names.)
#
# This script can only help to identify sources which need
# additional investigation per the advice of the HP documentation.
#
# Some programs that have dependencies on the size of node
# or host names will not be detected by this script

#
# Search for the obvious symbolic references.  Also search for
# “char” arrays sized by the literal equivalents for
# UTSLEN (9 or 8) and MAXHOSTNAMELEN (64).
#
grep –E –e uname –e utsname –e nodename   \
      –e SYS_NMLN –e SNLEN –e UTSLEN      \
      -e hostname –e MAXHOSTNAMELEN       \
      –e ‘char.*\[([89]|64)\]’     $@
```

Testing Tips

Any application enhanced to accommodate expanded node and/or host names should be subjected to the same qualification procedures as for any product update. In addition, the test procedures should set up an environment where the node and host names are set to long values.

Your network administrator should be able to provide a host/node name label of 63 bytes. The local network domain can be added (with dot delimiters) to form an FQDN. This results in an FQDN that is longer than 63 bytes and ensures that the host name string manipulated by the application exceeds the default limits.

You might want to set up a separate domain hierarchy with 255-63-1 = 191 total bytes, including delimiters (you'll need at least three levels). This allows testing with the maximum-length host name.

For many applications, however, setting up an environment with very long names does not require the attention of the network administrator. Before executing tests, the node and host names can be assigned arbitrary, made-up names. Then, after test execution, the original names should be restored.

Any long names used in testing should not have the same first 8 bytes as another full host name in the same domain. Using names whose first 8 bytes are unique ensures that any use of names truncated to 8 bytes do not allow successful name lookups (which can hide real errors).

Caution: Testing does not substitute for investigation. Testing can often miss problems that are easily identified by investigation. Testing should always be treated only as a final validation.

APPENDIX D: FLV Concepts and Usage

Function-Level Versioning (FLV) is a feature of the HP-UX compiler and linker that assists in the management of changes to interfaces. It provides the capability to define and export multiple versions of a function or data interface. To ensure correct operation, client references to the interfaces are bound to the desired version at link time.

HP-UX 11i v2 September 2004 Update, as well as later updates and releases, utilize this mechanism to provide two versions of the `uname()` function and, correspondingly, two versions of the associated `utsname` structure definition. The version used is selected during compilation.

In general, application code need not be aware of the two versions. The default compilation environment selects the version that is compatible with prior versions of HP-UX. To select the version that supports expanded node and host names, the option `–D_HPUX_API_LEVEL=20040821` is included in the compilation environment.

This appendix describes how the FLV feature works, how it implicitly affects interfaces that re-export versioned interfaces, how it is used for node names, and where it might need to be explicitly employed by library providers.

Caution: FLV is available for updating a small set of interfaces that are not extensively used. The feature has limitations that makes its widespread use ill advised. Its primary purpose is to enable HP to expand interfaces without requiring recompilation of applications. It is described here primarily for the benefit of developers whose interfaces re-export the node and host name-related interfaces. HP does not support use of FLV for other purposes.

Elements of FLV

This section describes the mechanisms and syntax of FLV.

A Simple Example

For the purposes of illustration, suppose the following interface has been defined for some time:

```
void logevent(struct eventinfo *evp);
```

The interface provides a general service used by many clients. However, because of the demands of the user community, a second parameter needs to be added. The interface is updated as follows:

```
void logevent(struct eventinfo *evp, void *eventdata);
```

This pleases the users who were demanding the enhancement. However, it creates problems for the users who have no need to provide data with the event. These users must add a second parameter, usually NULL. Worse, other users who are unaware of the change might re-link their application to the enhanced interface without having updated the source code. This can cause indeterminate behavior when `logevent()` attempts to de-reference the pointer that was not provided.

The interface providers might have been able to instead add a second `logevent_rev2()` interface to accept two parameters and leave the existing one alone. Defining a second name is preferred over using FLV. However, the change might have been dictated by revisions to standard specifications. A second name may not be acceptable (as is the case with `uname()` on HP-UX.)

The following subsections also refer to this example.

Source and Binary Names and the Version Attribute

Before proceeding with the description of FLV, note the following terms need. These are used when referring to interfaces exported from or imported into a compilation module.

source name
: The name of an interface as written in the program's source code file.

binary name
: The name of an interface as it appears in the symbol table of the program's (relocatable) object file, as used by the object linker to bind the external references of one object to another.

The source and binary names are usually the same. If the source program references the `logevent()` function, that name appears in the symbol table of its compiled object file as an undefined external function reference. For the source program that defines the `logevent()` function, that name appears in its object symbol table as an exported function.

With FLV, the binary name might be different from the source name because FLV adds a version identifier to the binary name. The version is specified as an attribute:

```
   void logevent(struct eventinfo *evp,
            void *eventdata) __attribute__((version_id("rev2"))) ;
```

The names of this log facility interface would be:

source name:	`logevent()`
binary name:	`logevent{rev2}()`

where `{rev2}` is the version identifier. This version identifier is applied to the symbol table entry of all object files that reference the function, as well as to that of the object file that defines the function. The significance of this is described in the next subsection.

Linking with Version Identifiers

The linker interprets the version identifier merely as part of the symbol name. As such, it requires that the whole name, including the version identifier, match between the reference and the definition before it can bind the reference.

Figure C-1 illustrates that an object that references the original, one-parameter, version of `logevent()`properly links to an object that defines the original version of that function. An object that references the new, two-parameter, version links to an object that defines that version. The converse is not true. It is not possible for `log.o` to satisfy the reference to `logevent{rev2}()` made by `clientB.o`. If only those two objects are passed to the linker, it will fail because the linker cannot resolve `logevent{rev2}().` Similarly, if only `clientA.o` and `log_rev2.o` are passed to the linker, it will fail to resolve `logevent().`

Note that if both `log.o` and `log_rev2.o` are combined into a single library object, that library can satisfy the external reference of either `clientA.o` or `clientB.o`.

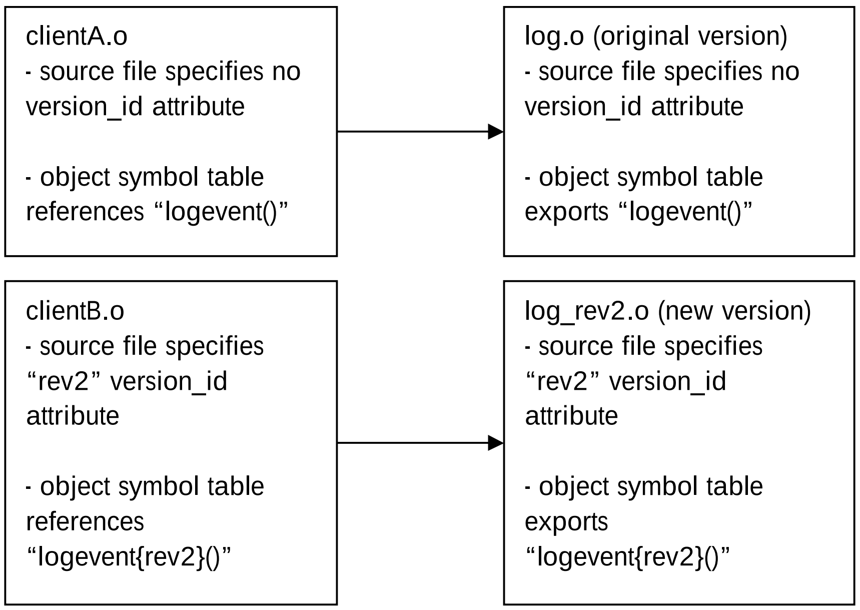
clientA.o
- source file specifies no version_id attribute
- object symbol table references "logevent()"
log.o (original version)
- source file specifies no version_id attribute
- object symbol table exports "logevent()"
clientB.o
- source file specifies "rev2" version_id attribute
- object symbol table references "logevent{rev2}()"
log_rev2.o (new version)
- source file specifies "rev2" version_id attribute
- object symbol table exports "logevent{rev2}()"

Figure D-1: Legal link binding with version tags

FLV for Data Definitions and Data Types

Interfaces can be both data structures and functions. A data structure can be defined and exported by one object and referenced (imported) by another.

The version attribute can be applied to data and types as well as to functions. Here are a few examples:

```
   struct __attribute__((version_id(“rev2”))) eventinfo { <fields> … };
   extern struct __attribute__((version_id(“rev2”))) logparams
                                                     logparmeters;
   int logstatus __attribute__((version_id(“rev2”)));
   typedef long logflags_t __attribute__((version_id(“rev2”)));
```

The following is a cleaner way to code this:

```
   #define REV2 __attribute__((version_id(“rev2”)))

   struct REV2 eventinfo { <fields> … };
   extern struct REV2 logparams logparmeters;
   int logstatus REV2;
   typedef long logflags_t REV2;
```

An object file generated from a program source containing the preceding declarations would exhibit the following binary names in its symbol table:

```
   logstatus{rev2}
   logparams{rev2}
```

Note that only exported or referenced scalar and structure names appear in the object's linker symbol table. By tagging the exported or referenced data with the version identifier, the linker ensures that objects bind to the desired data format.

Type and structure definitions do not appear in the symbol table, but the ability to apply a version identifier is important, as described in the next section.

Transitive API Changes and Inheritance

Consider the following structure, which defines a common data header:

```
struct std_hdr {
   int         hdr_identifier;
   size_t      hdr_data_size;
   time_t      hdr_creationtime;
   time_t      hdr_lastmodifytime;
};
```

It is included at the beginning of other structures, such as the following:

```
struct ds_a {                         struct_ds_b {
   struct std_hdr    a_hdr;              struct std_hdr    b_hdr;
   int               a_field1;           int               b_field1;
   int               a_field2;           int               b_field2;
}                                     }
```

Finally, these structures are used in some exported function interfaces:

```
int fetch_ds_a(char *name, struct ds_a *dsap);
int fetch_ds_b(int cmd, struct ds_b *dsbp);
int fetch_hdr(struct std_hdr *hdrp);
```

As an example, assume the above declarations have been previously exported. However, now the definition of `std_hdr` needs to change in order to accommodate a new field (for example, a new `time_t hdr_lastaccesstime` field). Because `std_hdr` is used in other data structure definitions and functions, those definitions and interfaces must also change. The change is said to be transitive. That is, the change propagates from that structure into function interfaces that include that structure, into other structures that include that structure, into function interfaces that include those other structures, and so on. A change in one structure can affect many interfaces.

The FLV mechanisms provide version inheritance. Given a modified type (which is tagged with a version attribute), the version identifier is propagated to any interface that uses the modified type. This avoids the need to explicitly add version attributes to every declaration and definition in the affected interfaces.

The following are detailed rules for version inheritance:

- For a structure or union, if one or more of its members have attached version identifiers, the compiler determines which of these version identifiers is lexically greatest. It records this high-water mark in its internal symbol table entry for the newly declared structure or union. For C++ classes, the compiler considers the base classes, nonstatic data members, and virtual member functions, but not static data members, static member functions, or nonvirtual member functions.

- For an array, pointer, or reference type, if the base type has an attached version identifier, the compiler associates the same version identifier with the derived type.
- For a C++ pointer-to-member type, if either the class or the base type has an attached version identifier, the compiler determines which of these version identifiers is lexically greater. It associates this high-water mark as the version identifier for the type.
- For a typedef declaration, if the base type has an attached version identifier, the compiler associates the same version identifier with the new type name.
- For a global variable declaration or definition, for a C++ namespace data member, or for a C++ class static data member, if its type has an attached version identifier, the compiler records the version identifier as part of the binary name for the variable.
- For a function declaration, definition, or function type, if its return type has an attached version identifier, of if one or more of its parameter types have attached version identifiers, the compiler determines which of these version identifiers is lexically greatest. It records the high-water mark as part of the binary name for the function. For C++ nonstatic member functions, including constructors, destructors, and operators, the compiler considers a version identifier attached to the class (because of the implicit `this` pointer parameter).
- If a version identifier is directly applied to a declaration that also inherits a version identifier through the above rules, the directly applied version identifier takes precedence, but it must not be lexically less than the inherited one.
- When a name is redeclared in a source file, the following restrictions apply:
  - If the first declaration has a version identifier, a subsequent declaration or definition must have the identical version identifier.
  - If the first declaration does not have a version identifier, a subsequent declaration or definition must not have a version identifier.

To summarize, the rules state that any definition that includes some other definition inherits the version from that other definition. Thus, a version identifier automatically becomes part of the binary name of every interface affected by the changed type (as long as an explicit version identifier was applied to that type). Also, when there are multiple version identifiers, the greatest one is inherited.

The inheritance of the version identifier means that developers need not explicitly apply version attributes to each and every affected definition and declaration. It is sufficient to apply the version at the point the change is made. This propagates to all affected interfaces.

Limitations of FLV

FLV does not manage every aspect of interface change because of some serious limitations. Due to these limitations, FLV should be used only when absolutely necessary. It should be limited to a small set of interfaces that are not extensively used. Other approaches should be exhausted before resorting to FLV.

The following subsections describe limitations of FLV.

Constants Have No Version

Version identifiers cannot be applied to symbolic or literal constants. If a constant is used in the definition of data structures or function parameters, the version attribute must be explicitly applied to all those definitions.

Dynamic Library Loading: shl_findsym(), dlsym()

Utilities that reference the symbol table of an object or library do not have access to source names. They have access only to the binary names. Thus, when looking up a function or data definition, the binary name must be supplied.

Debuggers

The debugger does not have access to source names. It has access only to the binary names. Thus, displays (such as stack traces) provide binary names. When looking up functions or data, the binary name must be supplied. Note also that a breakpoint set in one version of a function does not affect calls to another version.

Assembly and Fortran: No Version

The assembler and the Fortran language ignore the version attributes. The binary names are the same as the source names. These languages always reference the original version of an interface unless the source code is modified explicitly to use the binary name.

Incomplete Types

A pointer to an incomplete type might be used in another type (for example, as a structure member or function parameter). If the referenced type has a version identifier, that version identifier might not propagate to the pointer type or to any interfaces that use it. The workaround is to provide the complete type or apply a version identifier directly to the pointer.

Type Casts

Type casts in a program source might be used to cast a pointer to the wrong version of a type. The compiler checking scheme might fail to detect a mismatch or might diagnose false mismatches.

Extensive Use of Interfaces

If an interface is versioned that is used by the majority of client applications, or if many interfaces are versioned (perhaps through version inheritance), it can become impossible for client programs to effectively use the original interface versions. All client applications are, for practical matters, forced into using the new versioned interfaces. In this case, it is more appropriate to require recompilation of all client programs (with possible source code updates) to use the new versions of the interfaces and to not use FLV at all.

Versioning is Cumulative

Interfaces cannot be versioned independently; versions must be cumulative. A client application cannot individually pick which versions are used for each interface it uses. This constraint is due to the fact that structures or functions that re-export multiple versioned interfaces inherit the version identifier that is lexically greatest (the high-water mark).

Virtual Member Functions and Derived Classes

If a base class is recompiled with a new version identifier but the derived class is not, a call to that function on an instance of the derived class might not work properly. This situation is really a general C++ binary compatibility issue and can occur even in the absence of the FLV mechanism.

Defeats Single-Binary / Multiple-Release

Many software developers build their products on an earlier HP-UX release (for example, 11.00 or 11i versions 1.0, 1.5, 1.6) and deploy the single binary on that and all later releases and versions. The FLV mechanisms cannot be retrofitted to those releases. The same is true for the versioned interfaces themselves. Furthermore, because FLV is a compile-time feature, it is not straightforward to make run-time decisions about whether to use the original or new interfaces. So run-time conditional execution is not practical.

Fortunately, for node names, it is possible to switch to using the host name with a large buffer and avoid using the versioned `utsname` structure and `uname()` function, thereby avoiding use of the FLV mechanisms. For a description of this approach see “Building for Execution on Multiple Releases.”

Providing Multiple Interface Versions in a Library

FLV enables a library provider to export multiple versions of the same interface. This is done by including in the library two object files. One exports the original version of the interface, the second exports an enhanced version, with a version identifier in the binary name. (In theory, any number of different versions can be supplied in this way.)

It is not possible to provide multiple versions of an interface in a single program source file. This is because the compilation environment cannot handle multiple versions of a definition.

The developer must be careful to not export any other symbols from these compiled objects. Multiple objects exporting the same binary names can cause ambiguity when the linker must resolve references.

The following sections describe two general approaches to supplying multiple versions of a function interface.

Parallel Implementation

In a parallel implementation, the modules that provide the original and enhanced versions of an interface are completely separate and do not interact. They might use common supporting functions. In some cases, to the modules can have a common source, but that source is used to generate multiple objects.

As an example, consider again the `std_hdr` structure and the `fetch_hdr()` function from a previous example in this appendix. The following program source is called `fetch_hdr.c`:

```
#include “std_hdr.h”
int fetch_hdr(struct std_hdr *hdrp) {
       hdrp->hdr_identifier = get_header_id();
       hdrp->hdr_data_size = get_data_size();
       hdrp->hdr_creationtime = get_creation_time();
       hdrp->hdr_lastmodifytime = get_modify_time();
#ifdef NEWVERS
       hdrp->hdr_lastaccesstime = get_access_time();
#endif
       return(0);
}
```

where the `std_hdr.h` header has:

```
#ifdef NEWVERS
#define ATTR __attribute((version_id(“rev2”)))
#else
#define ATTR  /* no version identifier if not the new version */
#endif

struct ATTR std_hdr {
 int         hdr_identifier;
 size_t      hdr_data_size;
 time_t      hdr_creationtime;
```

```
     time_t      hdr_lastmodifytime;
    #ifdef NEWVERS
     time_t      hdr_lastaccesstime;
    #endif
    }
```

This single source file can be compiled into two object files by using the following commands:

```
    cc –c fetch_hdr.c;        mv fetch_hdr.o fetch_hdr_orig.o
    cc –DNEWVERS –c fetch_hdr.c;   mv fetch_hdr.o fetch_hdr_new.o
```

The result is that `fetch_hdr_orig.o` exports `fetch_hdr()`, and `fetch_hdr_new.o` exports `fetch_hdr{rev2}()`. Both object files and their functions can be combined into a common library so that clients built for either interface version can link properly.

Original Interface as Thin Converter

The parallel implementation approach is not always practical. It might not be reasonable to factor the accesses to the structure into a small routine. An alternative approach is to build one interface as a thin conversion wrapper around the other. This takes a bit of tricky typecasting. Consider the following source code (using the same `std_hdr.h` header file as in the preceding example):

```
  fetch_hdr_orig.c:
    /* Thin wrapper to provide nonexpanded version of fetch_hdr() */
    /* compile without –DNEWVERS, so binary name is fetch_hdr() */
    #include “std_hdr.h”
    int fetch_hdr(struct std_hdr *hdrp) {
     struct std_hdr_orig;  /* incomplete definition */
     extern int fetch_hdr_translate(struct std_hdr_orig *);
     return( fetch_hdr_translate( (struct std_hdr_orig *)hdrp ) );
    }

  fetch_hdr_translate.c:
    /* Translate expanded std_hdr structure to nonexpanded format */
    #define NEWVERS
    #include “std_hdr.h”
    /* define std_hdr_orig as original layout */
    struct std_hdr_orig {
     int         ohdr_identifier;
     size_t      ohdr_data_size;
     time_t      ohdr_creationtime;
     time_t      ohdr_lastmodifytime;
    }
    int fetch_hdr_translate(struct std_hdr_orig *ohdrp) {
     struct std_hdr nhdr;  /* local buffer for new header */
     retval = fetch_hdr(&nhdr);  /* calls new version of function */
     /* copy fields to original layout */
     ohdrp->ohdr_identifier = nhdr->hdr_identifier;
     ohdrp->ohdr_data_size = nhdr->hdr_data_size;
     ohdrp->ohdr_creationtime = nhdr->hdr_creationtime;
     ohdrp->ohdr_lastmodifytime = nhdr->hdr_lastmodifytime;
     return(retval);
    }
```

Here, the bulk of the underlying work is done using the enhanced data-structure version. It is converted to the original format just before returning it to the client.

Multiple Data Interfaces

The preceding discussion concentrated on function interfaces. Another possibility is that a data structure exported by a library is accessed directly by client code that uses the library.

For data that is read-only by clients, the general approach to be used by the provider is to update both the original and enhanced versions of the data structure at the same time (where the two have common fields).

For data that is potentially written by clients, all code that reads or writes the data must be operating on the same version. The recommendation is that only one version of that structure (possibly the expanded version) be supported. The FLV mechanism helps to ensure that clients reference only the same version of the data structure as the library exports.

FLV Applied to HP-UX Node Name Interfaces

This section describes how HP-UX 11i uses FLV to provide multiple versions of the node name interfaces.

FLV is used for node name interfaces only, not host name interfaces.   For the node name interfaces, version identifiers are assigned to the utsname structure and the uname() function. No other interfaces have version identifiers.

The setuname(), sethostname(), and gethostname() functions all accept a size parameter to specify how many significant bytes are being passed in or can be accepted. These functions can handle original sizes or new sizes; there is no need for separate versions.

The SYS_NMLN, SNLEN, UTSLEN, and MAXHOSTNAMELEN interfaces are all constants. FLV cannot apply version identifiers to constants.

The sys/utsname.h header file defines the node name-related constants, the utsname structure, and the uname function. This header file depends on the header stdsyms.h to process the _HPUX_API_LEVEL definition and, in turn, define the _INCLUDE_HPUX_API_LEVEL symbol to be used in other header files. The HP-UX sys/utsname.h header file, in simplified form looks like:

```
#include <sys/stdsyms.h>
#if _INCLUDE_HPUX_API_LEVEL >= 20040821
#define SYS_NMLN 257
#define SNLEN 257
#define UTSLEN 257
#define ATTR __attribute((version_id("20040821")))
#else
#define SYS_NMLN 9
#define SNLEN 15
#define UTSLEN 9
#define ATTR /* no version */
#endif

struct ATTR utsname {
   char      sysname[SYS_NMLN];
   char      nodename[SYS_NMLN];
   char      release[SYS_NMLN];
   char      version[SYS_NMLN];
   char      machine[SYS_NMLN];
```

```
        char      idnumber[SNLEN];
    #   if _INCLUDE_HPUX_API_LEVEL >= 20040821
        char      reserved1[SYS_NMLN];
        char      reserved2[SYS_NMLN];
    #   endif
    }

    extern int uname(struct utsname *) ATTR;
```

If an application is built with no special options, it has the original layout for the `utsname` structure, and the binary name of the `uname()` function is `uname()`. Applications that are enhanced for the expanded node name are built with the compiler option `–D_HPUX_API_LEVEL=20040821`. This option selects the expanded `utsname` structure layout (along with two additional fields reserved for future use), and the binary name of the function is `uname{20040821}()`.

Note also that the symbolic constants have different values based on the compilation mode. These values are consistent with the structure definition. The same conditional compilation is applied to the `MAXHOSTNAMELEN` constant in another header file.

The `libc` library contains the `uname()` and the `uname{20040821}()` functions. These are built into two separate objects from a common source.

Re-export of utsname Structure

Consider the following interface:

```
    struct xyz {
     int   xyz_field1;
     struct utsname xyz_field2;
    }
    extern int get_xyz(struct xyz *);
```

If this interface is exported outside the product, to client application code, it is said to re-export the `utsname` interface.

Interface Inherits Version

When the code that implements the preceding interface is rebuilt with the `–D_HPUX_API_LEVEL=20040821` compiler option, the structure `xyz` and the function `get_xyz()` inherit the version identifier `20040821`. Thus, only client programs that are also built with the `–D_HPUX_API_LEVEL=20040821` option can bind to the expanded `get_xyz()` function. Conversely, if the code that implements this interface is not rebuilt, only client programs that are not built with the `–D_HPUX_API_LEVEL=20040821` option can bind to it.

While the version is implicitly inherited and exhibited in the binary name for this structure and function, it might be a good idea to explicitly add the version attribute. This can improve clarity for any developers attempting to read and understand the program source code.

Options for Interfaces Which Re-export the utsname Structure

Developers have three choices when an interface re-exports the `utsname` structure:

- Support only the original version, build without expanded API level.
- Support only the expanded versions, build with `–D_HPUX_API_LEVEL=20040821` option.
- Support both versions.

The approach taken is entirely dependent on the flexibility or demands of those who maintain the client programs.

If the node name is not considered to be important to the interface (even though the whole `utsname` structure is included) it might be tempting to support only the nonexpanded version. One drawback to this is that client programs might want to use `utsname` structure and `uname()` function elsewhere in their program to support long node names. Although it might be manageable for the program to separate the two sets of references, it might cause some inconvenience. The converse situation might arise if only the expanded version is supported by the interface.

Supporting both versions of an interface provides the most flexibility to client programs. The manner in which multiple versions are provided depends on the design of the code that provides the `get_xyz()` interface. Either a parallel implementation approach or a thin converter might be practical. See "Providing Multiple Interface Versions in a Library".

Re-export of Node/Host Name Constants

Consider the following interface:

```
struct uvw {
 int   uvw_field1;
 char  uvw_host[MAXHOSTNAMELEN];
}
extern int get_uvw(struct uvw *);
```

If this interface is exported outside the product, to client application code, it is said to re-export the `MAXHOSTNAMELEN` interface. The descriptions in the following sections can also be applied in a similar fashion to interfaces that re-export the `SYS_NMLN, SNLEN`, and `UTSLEN` constants.

Interface Does Not Inherit Version from Constants

The major issue with re-exporting constants is that the FLV mechanism does not provide version identifiers. Thus, the `uvw` structure inherits no version if its definition is based on the expanded version of `MAXHOSTNAMELEN`. No link-time checking is done to ensure that clients and interfaces have compatible data layouts.

Because of this, it is very important that library providers scan all interface definitions for instances of any of the node and host name-related constants. If expanded node and host names are to be supported, the affected structure and function interfaces must be given an explicit version attribute.

Options for Interfaces Which Re-export Node/Host Name Constants

The options for constants are similar to those for re-export of the `utsname` structure:

- Support only the nonexpanded version, and build without expanded API level.
- Support only the expanded versions, and build with `–D_HPUX_API_LEVEL=20040821` option.
- Support both versions.

The approach taken is entirely dependent on the flexibility or demands of those who maintain the client programs. It is easiest to support only one version. However, supporting both versions provides the most flexibility to client programs.

In this case, the last two options will require explicit version attributes.

Explicit Version Attributes

Because interfaces that re-export constants do not inherit a version identifier, it is important to add one explicitly. The version attribute macro defined in `sys/stdsyms.h` can be used for this purpose. For example:

```
#include <sys/stdsyms.h>
#if _INCLUDE_HPUX_API_LEVEL >= 20040821
#  if MAXHOSTNAMELEN != 256
   error MAXHOSTNAMELEN and API level mismatch /* sanity check */
#  endif
#define UVW_ATTR _HPUX_API_VERS_20040821_ATTR /* version attribute */
#else
#define UVW_ATTR /* no version attribute */
#  if MAXHOSTNAMELEN != 64
   error MAXHOSTNAMELEN and API level mismatch /* sanity check */
#  endif
#endif

struct UVW_ATTR uvw {
 int   uvw_field1;
 char  uvw_host[MAXHOSTNAMELEN];
}
extern int get_uvw(struct uvw *) UVW_ATTR;
```

Note that the function `get_uvw()`, because it has a versioned type as a parameter, automatically inherits the version identifier from `struct uvw`. An explicit attribute is not required but is added for clarity.

The sanity checks provide some compile-time checking that the declarations have been laid out consistently with the OS headers.

Constants: Providing Multiple Versions

The manner in which multiple versions are provided depends on the design of the code that provides the `get_uvw()` interface. Either a parallel implementation approach or a thin converter might be practical. See “Providing Multiple Versions in a Library”.

With re-export of constants, it might be necessary to add explicit version attributes in the supporting code.

Glossary

11i v2 Update

Bundled updates to HP-UX 11i v2. Included in 11i v2 September 2004 Update is pre-enablement for expanded node name and host name. This includes the compiler versions which support FLV, and the header, kernel, and libc support of expanded node name (using FLV) and host name. It also includes updates to many commands and utilities to accommodate the long names. The remaining enablement for 11i V2 is provided in the optional NodeHostNameXpnd product bundle.

11i v3

The full release of HP-UX that follows the v2 release. It includes all enablement for expanded node and host names in the base OS product.

11.23

This is the "`release`" corresponding to HP-UX 11i v2 as provided by the `uname -r` command.

11.31

This is the "`release`" corresponding to HP-UX 11i v3 as reported by the `uname -r` command.

20040821

This is the optional API level which supports the expanded versions of the node name and host name interfaces. It is used as the value for the compilation symbol `_HPUX_API_LEVEL`. API levels are cumulative. Any future API level will be numerically greater than 20040821 and will include all preceding interface versions.

binary name

The name of an interface as it appears in the symbol table of the program's (relocatable) object file, as used by the object linker to bind the external references of one object to another.

Domain Name System (DNS)

The standard distributed host name lookup service for Internet host names (described in RFC 1034). DNS requires that individual host-name labels not exceed 63 octets (bytes), and that the entire host name (all labels and dot delimiters combined) does not exceed 255 octets.

EOVERFLOW

The error number returned from the nonexpanded version of the `uname(2)` system function when the value corresponding to any field exceeds the capacity of that field.

`expanded_node_host_names`

The kernel tunable that activates the expanded node name and host name capability. When turned on this tunable allows the system administrator to set large node and host name values. The default value on HP-UX 11i v2 and v3 is 0 (off). Programs can use the expanded APIs regardless of whether or not this tunable is turned on.

expanded node/host name clean

An application is said to be "expanded host/node name clean" if it uses no host or node name related interfaces, or has been rebuilt to use the expanded interfaces for the names.

fully qualified domain name (FQDN)

Internet host names can be either a single system name label such as “myhost”, or include the hierarchical Internet domain labels, such as “myhost.corp.hp.com”. The latter is a fully-qualified domain name. The full name is limited to 64 octets (bytes) on HP-UX versions prior to 11i v2 (and any 11i v2 system with the tunable `expanded_node_host_names==0` or not available). Name labels are limited to 63 octets by DNS.

Function-Level Versioning (FLV)

A feature of recent HP-UX compilers, linkers, and loaders that allows versioning of individual functions in a library. Previously, if any function in a library was versioned, the entire library must be versioned.

`gethostname(2)`

The API by which applications obtain the Internet host name for the system on which they are executing. See also node name, and *`MAXHOSTNAMELEN`*.

host name

The Internet name for the system. The host name can be a single name label or an FQDN. In most cases, customers set the host name to just the first name label (no domain qualification). See also DNS.

`hostname(1)`

A command that lets users or scripts obtain the Internet host name for the system on which they are executing. This command also supports the administrative setting of the Internet host name (via `sethostname()`). See also DNS.

`_HPUX_API_LEVEL`

This is the compilation symbol used to select the API version.

investigate

In this document, refers inspection of the source code and documentation. The investigation can be aided by the use of scripts to search for certain symbols or patterns.

long node name or host name

Host or node names that are longer (that is, have a greater string length) than the default maximums of 8 or 64 bytes, respectively.

`MAXHOSTNAMELEN`

A symbolic constant parameter that defines the maximum length of host name strings. Its value is 64 for the default compilation environment. Its value is 256 if the 20040821 HP-UX API level is selected. It is often used to establish the size of buffers that accept the returned value from `gethostname(2)`. However, its use is discouraged. It is better to use `sysconf(_SC_HOST_NAME_MAX)`.

NodeHostNameXpnd

The optional product bundle for HP-UX 11i v2 that allows you to set the node or host name longer than the prior limits. For 11i v3 or later updates and releases the capabilities of this bundle are included in the base OS product.

node name

This is the system’s UUCP name, separate from the Internet host name. HP documentation strongly recommends that the node name be the same as the first label of the host name. Many products assume the node name can be used as the Internet name for the system. The node name can be accessed by the uname(1) command or by the uname(2) system function.

nodename, utsname.nodename

The nodename is the field within the utsname structure as returned by the uname(2) system function. For programs compiled with _HPUX_API_LEVEL=20040821, it can be up to 255 bytes (plus null terminator).

re-export of interfaces

The definition of an exported interface (A) can include part of some other interface definition (B). In this case A "re-exports" B. For example, consider the interface myfunction() which returns a structure mystruct which contains the field myhostname char [MAXHOSTNAMELEN]. The interface myfunction() re-exports part of the host name interface, specifically, the MAXHOSTNAMELEN interface.

RFC 1034

The Internet specification for the Domain Name System (DNS). A copy can be found at (http://www.rfc-archive.org/getrfc.php?rfc=1034).

_SC_HOST_NAME_MAX

A parameter to the sysconf(2) system function by which a program can obtain the maximum size of a host name.

sethostname(2)

The API used by system administrators to set the Internet host name for the system. The name can later be retrieved by applications via the gethostname(2) API or the hostname(1) command. See also node name and *MAXHOSTNAMELEN*.

setuname(2), setuname(1m)

The API and command, respectively, used by administrators to set the UUCP node name for the system. The name can later be retrieved by applications via the uname(2) interface.

SNLEN

See *SYS_NMLN*.

source name

The name of an interface as written in a program's source code file.

SYS_NMLN, SNLEN, UTSLEN

Symbolic constants that specify the length of the fields in the utsname structure. . All three have the value 257 in compilation environments when the 20040821 HP-UX API level is selected. In the default compilation environment, the value is 9 (SYS_NMLN and UTSLEN) or 15 (SNLEN).

uname(2), uname(1)

The API and command, respectively, used to obtain a set of system parameters, including the node name. The node name can be obtained from the utsname.nodename field returned by the API, or from the command using the –n or –a option. There are two versions of the uname(2) API – a nonexpanded version compatible with previous versions of HP-UX, and an expanded version. The version used by a program is determined by the setting of the _HPUX_API_LEVEL compile-time symbol.

`uname_eoverflow`

The kernel tunable parameter that controls whether the EOVERFLOW error is reported by the `uname(2)` system function. The default value on HP-UX 11i v3 is 1 (on).

`UTSLEN`

See *`SYS_NMLN`*.

`utsname`

The program data structure that is returned by the `uname()` API. It contains the `nodename` field. See also *`SYS_NMLN`*, *`SNLEN`*, and *`UTSLEN`*.

UUCP

UNIX to UNIX file copy. UUCP is a pre-Internet protocol and set of commands to support transfer of files. It appears as part of some standards.

Part Number: 5991-6425, January 2007