UNIX, Linux to OpenVMS Porting Guidelines

Table of Contents

1. Introduction

This document describes the process to port applications from UNIX or Linux systems to OpenVMS systems. It also provides information about the issue, typically encountered while porting and also workarounds for these issues. The objective of this document is to leverage the knowledge and experience of the OpenVMS community and to be a source of information for the developers who intend to port applications to OpenVMS.

2. Intended Audience

This document is intended for developers who intend to port applications from UNIX or Linux systems to OpenVMS system.

3. Planning and Preparation

The planning and preparation phase of the porting process consists of the following steps:

3.1. Data Collection and Assessment

3.1.1. Pre-requisites and Dependencies

This step involves identifying all the pre-requisites and compile-time or runtime dependencies for the application on the source platform. These pre-requisites or dependencies may require licenses. Most of the compilers available for OpenVMS also require licenses. The license requirements for all the required applications or tools must be determined as a part of the identification process.

OpenVMS has a number of open source and third-party libraries and applications. The following links can be used to check for the availability of the required applications on OpenVMS systems:

- Partner Applications
  http://www.hp.com/go/openvms/solutions

- Open Source/Freeware Applications and Tools
  http://h71000.www7.hp.com/opensource/opensource.html
  http://h71000.www7.hp.com/openvms/freeware/
  http://de.openvms.org/OpenVMS-Ports/

3.1.2. Analyzing the Source Code Layout and Building the Application

This step involves understanding the source structure and the build system for the application to be ported. An understanding of the environment required for the build as well as the various flags used will help speed up the porting process.

3.1.3. Creating the Source Package

The complete source package for the application to be ported needs to be identified. This includes all the compile-time or run-time dependencies. It is recommended to create an archive for the source package and build the application using the archive on the source platform. This ensures that the archive created is proper and can be used on the target platform. It also makes the process of comparing results between the source and target platforms much easier as the same source package is used at both instances.

3.2. Setting up the Porting environment

The porting environment must have the GNV software installed, the required compilers, scripting languages, development tools, and the system parameters must be set.

The gcc and g++ compilers provided as a part of GNV, are wrappers around the native HP C and C++ compilers. Hence, the HP C and C++ compilers must be installed and licenses must be loaded for the gcc and g++ wrappers to work.

3.2.1. System parameters

The following table provides the minimum system quota settings required for a porting environment.

Channelcnt	4096
UAF Fillm	4096
Wsdef	2048
Wsquo	4096
Wsextent and Wsmax	16384
Pgflquo	2097152*
bytlm	400000
biolm	150
diolm	150
tqelm	100
Channelcnt	4096
UAF Fillm	4096
Wsdef	2048

* Indicates the appropriate number for Pgflquo (2 x heap-size).

For example, 128 MB (2*128*1024*1024)/512 = 524288. When you increase the Pgflquo parameter, you should always increase the system's page file size to accommodate the new Pgflquo parameter, if required.

3.2.2. GNV

OpenVMS delivers UNIX tools and utilities using the GNU's not VMS (GNV) software package. GNV is a GNU's not UNIX (GNU) based UNIX shell and utilities environment for OpenVMS, which implements the UNIX BASH shell (Bourne Again Shell) and provides many UNIX tools and utilities for:

- General purpose
- Command manipulation
- Program creation
- File manipulation
- Text processing
- Printing
- Networking

For the latest version of GNV, see the web address at:

http://h71000.www7.hp.com/opensource/gnv.html

The GNV "Read Before Installing" guide contains the detailed installation instructions and is available at:

http://h71000.www7.hp.com/opensource/gnvreadme_first.html

Recommendations for using GNV:

- ODS-5 disks are recommended for porting activities. While GNV installation requires ODS-5, user disks may be ODS-2 disks. These disks must be converted to ODS-5 before porting an application.
- Extended parsing must be enabled to support long file names and special characters in filenames. Execute the following command to enable long filenames and special characters:

```
$ SET PROCESS/PARSE_STYLE=EXTENDED
```

- To prevent bash from "fallback" to DCL for command execution, the following command must be executed in bash:

```
bash$ export GNV_DISABLE_DCL_FALLBACK=1
```

  This is useful for the "configure" scripts supplied with the open source packages.

- DECC$PIPE_BUFFER_SIZE must be set to 65000, because of the bash pipe handling mechanism. To set the buffer size, execute the following:

```
$ DEFINE/JOB DECC$PIPE_BUFFER_SIZE=65000
```

- Enable DECC$STREAM_PIPE for UNIX compatible stream I/O by defining the command as follows:

```
$ DEFINE/JOB DECC$STREAM_PIPE ENABLE
```

3.2.3. General Environmental Differences between GNV/OpenVMS and UNIX

This section discusses some of the differences observed between the GNV features and utilities on OpenVMS and their counterparts on UNIX systems.

Root Directory

OpenVMS systems do not have a single root directory similar to that of UNIX systems. The UNIX root directory (/) is the top level of the system file hierarchy. All directories on the system, irrespective of the physical device are located under the root. On an OpenVMS system, the closest entity to the UNIX root directory is the top-level directory of a specific device. The character "/" is not recognized as a directory.
OpenVMS Version 8.3 has a new feature in the SET ROOT command to specify the location of the root directory. GNV points this root at the primary GNV directory, generally on the system disk. Furthermore, GNV creates a number of directories commonly found immediately under a UNIX root directory: /etc , /usr , /bin , /lib ,. and so forth.
This root directory is the top level of the GNV directory tree. You may use it to locate numerous files and directories. However, unlike a UNIX system, it is not true that all files and directories in the system can be found under the root. To accomplish that, the GNV utility mnt is used to connect all mounted disks to directory entries under the /mnt directory.

Multiple Versions of a File

OpenVMS operating systems maintain multiple versions of a file, with the highest version number being the most recent. UNIX maintains only the most recent version of a file. With a few exceptions, GNV supports this UNIX feature. For example, the rm utility removes all

versions of a file. Some of the utilities that still act only on the most recent version of a file, leaving the earlier versions in place are:

- mv
- chmod
- chown
- ln

For example, if you use mv to move (or rename) a file, only the highest version number file is moved. The lower versions (older) files are left in place.

Case Sensitivity in File Names

Normally, OpenVMS systems are not case sensitive. However, on ODS-5 devices you can enable case sensitivity for file names by using the following command at the OpenVMS DCL prompt or in a login command file:

```
SET PROCESS/CASE=SENSITIVE/PARSE_STYLE=EXTENDED
```

File Names Beginning with a Period

OpenVMS lets you create a file name beginning with a period. The OpenVMS DIRECTORY command will list such files. UNIX systems consider such files as hidden. The UNIX ls command does not list such files unless, for example, the -a option is used or the file name is specified in the command line.

4. Development Guidelines

This section describes the development guidelines required for porting applications to OpenVMS systems.

4.1. Data Structures and Data types

4.1.1. int64_t and uint64_t

In UNIX or Linux, int64_t and uint64_t data types are defined in the stdint.h, sys.h or types.h header files. However, in OpenVMS, these data types are defined in the inttypes.h header file. Hence, the header file inttypes.h must be included when porting to OpenVMS systems.

4.1.2. Mismatch issues

Most of the UNIX data structures available on OpenVMS are POSIX compliant and contain all the members as required by the standards. However, many Open Source projects use structure members, which are part of extensions to standards.
For example, the UNIX `passwd` structure is defined as shown:

```
struct passwd
{
   char *      pw_name;
   char *      pw_passwd;
   uid_t       pw_uid;
   gid_t       pw_gid;
   char *      pw_gecos;
   char *      pw_dir;
   char *      pw_shell;
} ;
The OpenVMS passwd structure is defined as given below.
struct passwd
```

```
{
   char *      pw_name;
   uid_t       pw_uid;
   gid_t       pw_gid;
   char *      pw_dir;
   char *      pw_shell;
} ;
```

The members pw_passwd and pw_gecos in the UNIX passwd structure are extensions to the standard.

If the application makes use of fields which are extensions to the standards, then the recommended approach is as follows:

- If it is possible to extract the required information using native OpenVMS APIs, a stub has to be written for the call where the OpenVMS C RTL call must be made for populating the standard members and the native call(s) for populating the extended members. The UNIX style structure, as expected by the application must then be passed back by the stub.
- If it is not possible to extract the required information on OpenVMS, then code using the extended members has to be conditionally compiled out for OpenVMS.

4.1.3. Checking for Endianness of Data

When porting from UNIX to OpenVMS, check the code for endianness of data.

For example, when accessing data structures, convert it from big-endian to little-endian before accessing it.

For example, to be inline with OpenVMS endianness, the cell physical location, CPU present and CPU de-configured data that is got from the ACPI data and IPMI is handled as follows:

```
_cellPhysicalLocation = SWAP_BYTES_64(_cellPhysicalLocation);
cpuPresent = SWAP_BYTES_16(cpuPresent);
cpuDeconfigured = SWAP_BYTES_16(cpuDeconfigured);
```

where, SWAP_BYTES_* macros are defined as follows:

```
#define SWAP_BYTES_64(value)  \
( ((uint64_t)(value) >> 56) | \
((uint64_t)(value) << 56) |   \
(((uint64_t)(value) >> 40) & 0x000000000000ff00) | \
(((uint64_t)(value) << 40) & 0x00ff000000000000) | \
(((uint64_t)(value) >> 24) & 0x0000000000ff0000) | \
(((uint64_t)(value) << 24) & 0x0000ff0000000000) | \
(((uint64_t)(value) >> 8) & 0x00000000ff000000) | \
(((uint64_t)(value) << 8) & 0x000000ff00000000)
)

#define SWAP_BYTES_32(value) \
(((uint32_t)(value) >>24) | \
((uint32_t)(value) <<24) | \
(((uint32_t)(value) >>8) & 0x0000ff00) | \
(((uint32_t)(value) <<8) & 0x00ff0000)
)

#define SWAP_BYTES_16(value) \
((((uint16_t)(value)) >> 8) | \
(((uint16_t)(value)) <<8 )
```

4.1.4. 32-bit and 64-bit Issues for Bit-wise Operations on Pointer Variables

When bit wise operations are applied on size_t pointer variables the results for 32 bit and 64 bit are varied. For 64 bit code, the pointer type must be typecast according to your requirement.

4.1.5. Stat() on OpenVMS

The default definition of the stat structure on OpenVMS does not have the members, `st_blksize` and `st_blocks`. To use the standard-compliant definition of stat structure and associated definitions, the application must be compiled with the _USE_STD_STAT feature test macro defined.

4.1.6. Case Sensitivity in Symbol Resolution

Symbol resolution on UNIX is case sensitive. However, symbol resolution on OpenVMS is not case-sensitive by default. To maintain case sensitivity in symbol names, the compiler flag `/names=as_is` must be used.

4.2. Troubleshooting GNV issues

Following are few troubleshooting techniques which can used to resolve the most commonly faced issues with GNV while setting up the build environment.

4.2.1. GCC and G++ utilities in GNV

The gcc and g++ commands in GNV are a wrapper that invokes the CXX compiler. The wrapper utility processes UNIX-style compile and link (and other related) commands and maps them to corresponding DCL commands on VMS.

4.2.2. GNV 'BIN' logical is not defined during the setup

During GNV setup, BIN logical is not defined. Hence, the following command fails

```
bash$ /bin/sh
BASH.EXE: /bin/sh: no such file or directory
```

To resolve this issue, define the following:

```
$ DEFINE BIN "GNU:[BIN]"
```

4.2.3. DCL Utility may ACCVIO when Invoked from bash shell

If the PATH is not set properly, DCL command at BASH prompt will try to invoke the VMS dcl utility instead of GNV's dcl usitility and will result in an accvio. To resolve this issue, define PATH to /gnu/bin instead of /bin because the /bin points to the sys$system.

```
bash$ export PATH=/gnu/bin:/GNU/BIN:/usr/bin:/usr/local/bin:.
```

4.2.4. GNV DCL Wrapper does not Return the UNIX Success Value

DCL commands do not return zero on success. Therefore, if you use them in a makefile, either instruct make to ignore the return value with "-" or invoke something after the DCL command that will return a zero on success, such as:

```
DCL DIR /FULL | grep MAKEFILE

OR

DCL DIR MYFILE.TXT ; dcl_return $?
where dcl_return is:
#!/gnu/bin/bash
# Converts DCL return value into bash return code
# In DCL, odd value means success. In bash, 0 value means success.
# If DCL failure (even), return the DCL return value.
if [ $(( $1 & 1 )) -eq "1" ]
then
    exit 0
else
    exit $1
fi
```

4.2.5. Make File Exceeds the Maximum Command Line Length

This issue occurs only when the SHELL variable is defined inside the makefile. Work around is to remove the shell variable so that make will be invoked from the default path, `/bin/sh`.

4.2.6. Disabbling DCL Message Display on BASH

While executing the configure script DCL messages may get displayed on the bash.

```
bash$ ./configure
%DCL-W-MAXPARM, too many parameters - reenter command with fewer parameters
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\
%DCL-W-IVKEYW, unrecognized keyword - check validity and spelling \-C\ %DCL-W-
MAXPARM, too many parameters - reenter command with fewer parameters
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\
```

To avoid DCL message on the bash, enable the symbol export,
GNV_DISABLE_DCL_FALLBACK=1

4.2.7. 'rm' may fail to Remove the Directory

While running the configure script `rm` may fail to remove the directory and displays the following message

```
+ core.conftest.* rm -f -r conftest* confdefs.h conf543176842
rm: cannot remove directory `conf543176842': directory not empty (This is a
typical VMS error message)
```

This is because the directory may contain the symbolic link file. Which will be not deleted using the `rm` utility on the bash. So under DCL, list and remove the symbolic link.
If the symbolic link is not present, the other workaround to this problem is to rename the conf.dir in the configure file to `conf.<something>`. Because the ".dir" extension is not properly handled.

```
bash> mv configure configure.org
bash>sed 's/conf$$\.dir/conf$$\.ddd/g' configure.org > configure
```

4.2.8. File not Found Error

While running the configure script, file not found error may occur because of improper file permissions. To avoid this, change the file permissions using `chmod` command, such as `chmod -R 777 *`

4.2.9. Handling Reserved Characters in Library Name such as tcpip$library:tcpip$lib.olb

The '$' in the name "tcpip$library" should be preceded by the '\' to work on bash. Also, here -l switch does not handle the special character, '$' in the library name properly. You can bypass the usage of -l switch and include the library into the PATH "-L'PATH'" as follows:

```
bash$ g++ -Os -o /bin/smtp -L/TCPIP\$LIBRARY/TCPIP\$LIB
```

4.3. File Operations

This section describes the file operations required for porting applications to an OpenVMS system.

4.3.1. Stream file sharing by single and multiple processes

In an UNIX or Linux system, if a single stream formatted file is opened by two processes, and if the first process modifies that file, the modification gets reflected in the second process and vice versa. This is due to sharing of the buffers in memory and the way the UNIX file system synchronizes access to these buffers.

In UNIX/Linux, Inode sharing between the different processes is available where one process writes and another process knows about the changes made by that process. This is possible in UNIX/Linux because each process has an open File table, which points to a common Inode table.

In OpenVMS, each process points to the same File Control Block. The UNIX file system internally manages locking in a way to make this possible. The OpenVMS file system also does this. However, this is currently not possible using C RTL for POSIX calls, which the UNIX applications would typically use.

For example, in case of a single process:

```
fd1=open("test.dat"...);
  write(fd1, "hello"...) ;

  fd2=open("test.dat"...);
  write(fd2, "goodbye"...);
  close(fd2)
  lseek(fd1, to the beginning);
  read(fd1...);
```

Here, the read using fd1 will get "hello". This requires fd1 to be closed and opened to read "goodbye". User can work around this issue using the following:
This workaround lets the user to open the file as many times as they would like.
However, all file I/O operations is done using a single file descriptor.

```
fd1 = open ("test.dat");
fd2 = open("test.dat");
```

fd1 is considered as the Master_fd. Fd1 allows to state the user calls, my_read(fd1,...)

`my_read()` does the following:

Lookup table of shared fd(s)

```
if (fd == master_fd)
 read(fd,...)
else {
   int savepos = lseek(master_fd, current_pos)
   lookup current position of fd2
   lseek to that position lseek(master_fd,...)
   read(master_fd,...)
   save and store the new position of fd2 = lseek(master_fd...)
   restore the file pointer to master_fd lseek(master_fd, savepos)
   return result from the read.
}
```

This works similarly with lseek, write, close and open calls

An alternative solution is to use the INDEX file format mechanism available natively in OpenVMS using RMS calls. If the file has to be opened in a shared mode for multiple reads, the DECC$FILE_SHARING logical has to be enabled. It sets the RMS flags (FAB$M_DEL | FAB$M_GET | FAB$M_PUT | FAB$M_UPD) as a logical OR with any sharing mode specified by the caller.

4.3.2. Symbolic Links and Hardlinks

OpenVMS provides symbolic links and hardlinks. Symbolic links are special files that point to another file. A symbolic link is a directory entry that associates a filename with a text string that is interpreted as a POSIX pathname when accessed by certain services. The path specified by a Symbolic link on OpenVMS is a POSIX path.

Hardlinks are also special files that point to another file. However, a key difference is that counters are maintained for the links. Thus, a file is deleted only when the last entry is deleted. Hardlinks need to be enabled for specific volumes.

4.4. POSIX Root and Mount Points

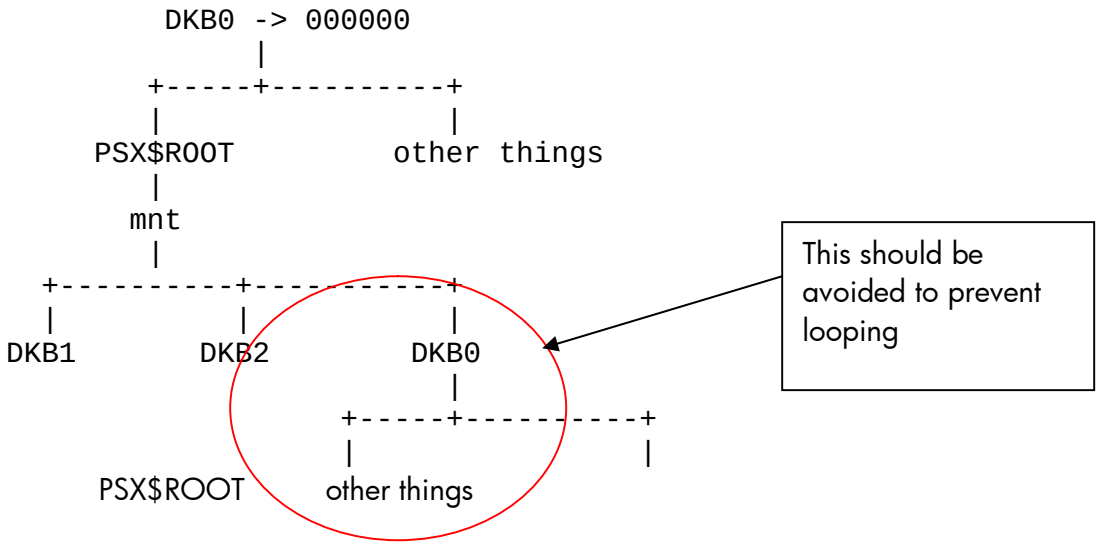
OpenVMS Version 8.3 and higher provides POSIX pathname processing support.
A mount point is a location where a file system is attached. The UNIX file system is essentially equivalent to the OpenVMS concept of a disk volume.

This poses some issues as the POSIX pathnames begin with "/" which points to the top level of the file system. On OpenVMS each disk volume has a top level directory of its own.

The concept of POSIX root is provided in OpenVMS to address this. Any directory can be designated as the POSIX root. All path resolutions use the POSIX root as the top level directory. Files that do not reside in the directory tree under the root directory will be inaccessible using a POSIX pathname. Note that this is the case only when POSIX-compliant

modes are enabled, that is, DECC$POSIX_COMPLIANT_PATHNAMES is defined to one of the allowed values.

However, designating any disk volume as the POSIX root makes files on other disk volumes inaccessible. The concept of mount points is provided for addressing this.
Other disk volumes can be mounted by means of a mount point under the POSIX root directory making them accessible. However, the disk volume containing the POSIX root should not be mounted under it as this could lead to a loop in directory tree traversals.



4.5. UNIX Portability Logicals

OpenVMS requires a number of C RTL logicals to be defined to set the environment for UNIX/LINUX applications to run. The following table lists the important logicals and their function.

DECC$FILE_SHARING	Provides the UNIX type of file locking behavior.
DECC$FILENAME_UNIX_REPORT	All file names are reported in UNIX style.
DECC$EFS_CASE_PRESERVE	Case is preserved for file names.
DECC$FILENAME_UNIX_ONLY	File names are never interpreted as OpenVMS style names
DECC$ARGV_PARSE_STYLE	Case is preserved in command line arguments.
DECC$FILENAME_UNIX_NO_VERSION	OpenVMS version numbers are not supported in UNIX style file names.
DECC$EFS_CHARSET	UNIX names can contain ODS-5 extended characters, support includes multiple dots and so on.
DECC$RENAME_NO_INHERIT	The new name for the file does not inherit anything such as the device, directory, file type, and version from the old file.
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION	Conversion routine decc$to_vms will only treat the first element of a UNIX style name as a logical name if there is a leading slash "/".
DECC$FILE_SHARING	Provides the UNIX type of file locking behavior.

DECC$FILENAME_UNIX_REPORT	All file names are reported in UNIX style.
DECC$EFS_CASE_PRESERVE	Case is preserved for file names.
DECC$FILENAME_UNIX_ONLY	File names are never interpreted as OpenVMS style names

These logicals are part of C RTL to ease the porting of the UNIX applications on OpenVMS. These logicals must be defined for UNIX kind of behavior, handle UNIX Style File Names and UNIX style file attributes on VMS.

The best method to define these logicals is by using Lib$initialize, which gets executed before the main() function. This provides an option to initialize the required logicals for the UNIX like behavior on VMS.

It is also possible to call functions of your own from lib$initialize

The following sample program illustrates the use of lib$initialize.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void init_logicals( void);
#pragma nostandard
        int lib$initialize();
        globaldef { "LIB$INITIALIZE" } readonly _align (LONGWORD)
                int spare[8] = {0};
        globaldef { "LIB$INITIALIZE" } readonly _align (LONGWORD)
                void (*x_my_init)() = init_logicals;
        /*
        ** Force a reference to LIB$INITIALIZE to ensure it
        ** exists in the image.
        */
        globaldef int (*lib_init_ref)() = lib$initialize;
#pragma standard
void init_logicals( void)
{
        printf("\nThis is before you called main");
        printf("\n I am in init  logicals");
}
int main()
{
        printf("\n \n I am in main");
}
$run init
This is before you called main
 I am in init  logicals
 I am in main
```

4.5.1. Setting UNIX Behavior Using a Single Logical

With the DECC$UNIX_LEVEL logical name, you can manage multiple C RTL feature logical names at once. The value you set has a cumulative effect: the higher the value, the more number of groups get affected. The principal logical names affecting UNIX like behavior are grouped as follows:

- 1 General corrections
- 10 Enhancements
- 20 UNIX style file names

- 30 UNIX style file attributes
- 90 Full UNIX behavior

Level 30 is appropriate for programs similar to UNIX, such as BASH and GNV. Setting a value of 20, for example, enables the entire feature logicals associated with DECC$UNIX_LEVEL of 20, 10, and 1.

The DECC$UNIX_LEVEL values and associated groups of affected feature logical names are:

General Corrections (DECC$UNIX_LEVEL 1)	
DECC$FIXED_LENGTH_SEEK_TO_EOF	1
DECC$POSIX_SEEK_STREAM_FILE	1
DECC$SELECT_IGNORES_INVALID_FD	1
DECC$STRTOL_ERANGE	1
DECC$VALIDATE_SIGNAL_IN_KILL	1
General Enhancements (DECC$UNIX_LEVEL 10)	
DECC$ARGV_PARSE_STYLE	1
DECC$EFS_CASE_PRESERVE	1
DECC$STDIO_CTX_EOL	1
DECC$PIPE_BUFFER_SIZE	4096
DECC$USE_RAB64	1
UNIX style file names (DECC$UNIX_LEVEL 20)	
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION	1
DECC$EFS_CHARSET	1
DECC$FILENAME_UNIX_NO_VERSION	1
DECC$FILENAME_UNIX_REPORT	1
DECC$READDIR_DROPDOTNOTYPE	1
DECC$RENAME_NO_INHERIT	1
DECC$GLOB_UNIX_STYLE	1
UNIX like file attributes (DECC$UNIX_LEVEL 30)	
DECC$EFS_FILE_TIMESTAMPS	1
DECC$EXEC_FILEATTR_INHERITANCE	1
DECC$FILE_OWNER_UNIX	1
DECC$FILE_PERMISSION_UNIX	1
DECC$FILE_SHARING	1
UNIX compliant behavior (DECC$UNIX_LEVEL 90)	
DECC$FILENAME_UNIX_ONLY	1
DECC$POSIX_STYLE_UID	1
DECCUSE_JPI_CREATOR	1
DECC$DETACHED_CHILD_PROCESS	1

4.6. Dynamic Shared Object method for VMS

HP OpenVMS CRTL routines dlopen, dlsym, dlclose can be used to allow sharable images to be loaded at the runtime, to return the address of the symbol name in the sharable image and unload the shared library. The dlclose function deallocates the address space allocated by the HP C RTL for the handle. There is no way on OpenVMS systems to unload a shareable image dynamically loaded by the LIB$FIND_IMAGE_SYMBOL routine, which is

the routine, called by the dlsym function. In other words, there is no way on OpenVMS systems to release the address space occupied by the shareable image brought into memory by dlsym.

4.7. Solutions for Fork Related issues

4.7.1. Creating the Child Process

The vfork and exec functions in the HP C RTL on OpenVMS systems work differently than on UNIX systems:

- On UNIX systems, vfork creates a child process, suspends the parent, and starts the child running where the parent left off.
- On OpenVMS systems, vfork establishes context later used by an exec function, but it is the exec function, not vfork, that starts a process running the specified program.

For a programmer, the key differences are:

- On OpenVMS systems, code between the call to vfork and the call to an exec function runs in the parent process. On UNIX systems, this code runs in the child process.
- On OpenVMS systems, the child inherits open file descriptors and so on, at the point where the exec function is called. On UNIX systems, this occurs at the point where vfork is called.

Enabling the DECC$DETACHED_CHILD_PROCESS feature logical allows child processes to be created as detached processes instead of subprocesses. This feature has only limited support. In some cases, the console cannot be shared between the parent process and the detached process, which can cause exec to fail.

The exec functions use the LIB$SPAWN routine to create the subprocess and activate the child image within the subprocess. This child process inherits the parent's environment, including all defined logical names and command-line interpreter symbols. By default, child processes also inherits the default (working) directory of their parent process. However, you can use the decc$set_child_default_dir function to set the default directory for a child process as it begins execution.

The exec functions tranmsit the following information to the child:

- The parent's umask value, which specifies whether any access is to be disallowed when a new file is created.
- The file-name string associated with each file descriptor and the current position within each file. The child opens the file and calls lseek to position the file to the same location as the parent. If the file is a record file, the child is positioned on a record boundary, regardless of the parent's position within the record.
- This information is sent to the child for all descriptors known to the parent including all descriptors for open files, null descriptors, and duplicate descriptors.
- File pointers are not transferred to the child. For files opened by a file pointer in the parent, only their corresponding file descriptors are passed to the child. The fdopen function must be called to associate a file pointer with the file descriptor if the child will access the file-by-file pointer

- The DECC$EXEC_FILEATTR_INHERITANCE feature logical can be used to control whether or not a child process inherits file positioning, and if so, for which access modes.
- The signal database. Only SIG_IGN (ignore) actions are inherited. Actions specified as routines are changed to SIG_DFL (default) because the parent's signal-handling routines are inaccessible to the child.
- The environment and argument vectors.

When everything is transmitted to the child, exec processing is complete. Control in the parent process then returns to the address saved by vfork and the child's process ID is returned to the parent.

Following programs demonstrates how to create a child process and communicate between parent and the child using the functions vfork and exec on OpenVMS:

Creating the Child Process

```
/* This example creates the child process. The only */
/* functionality given to the child is the ability to */
/* print a message 10 times. */
#include <climsgdef.h> /* CLI status values */
#include <stdio.h>
#include <perror.h>
#include <processes.h>
#include <stdlib.h>
static const char *child_name = "exec_image_child.exe" ;
main()
{
       int status,
       cstatus;

       /* NOTE: */
       /* Any local automatic variables, even those */
       /* having the volatile attribute, may have */
       /* indeterminant values if they are modified */
       /* between the vfork() call and the matching */
       /* exec() call. */
       if ((status = vfork()) != 0) {
              /* This is either an error or */
              /* the "second" vfork return, taking us "back" */
              /* to parent mode. */
              if (status < 0)
                     printf("Parent - Child process failed\n");
              else {
                     printf("Parent - Waiting for Child\n");
              if ((status = wait(&cstatus)) == -1)
                     perror("Parent - Wait failed");
              else if (cstatus == CLI$_IMAGEFNF)
                     printf("Parent - Child does not exist\n");
              else
                     printf("Parent - Child final status: %d\n", cstatus);
              }
       }
       else { /* The FIRST Vfork return is zero, do the exec */
              printf("Parent - Starting Child\n");
              if ((status = execl(child_name, 0)) == -1) {
                     perror("Parent - Execl failed");
                     exit(EXIT_FAILURE);
              }
       }
}
------------------------------------------------------------
/* This is the child program that writes a message */
```

```
/* through the parent to "stdout" */
#include <stdio.h>
main()
{
       int i;
       for (i = 0; i < 10; i++)
       printf("Child - executing\n");
       return (255) ; /* Set an unusual success stat */
}
```

Passing Arguments to the Child Process

```
/* In this example, the arguments are placed in an array, gargv, */
/* but they can be passed to the child explicitly as a zero- */
/* terminated series of character strings. The child program in this */
/* example writes the arguments that have been passed it to stdout. */
#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>
const char *child_name = "childarg_child.exe" ;
main()
{
       int status,
       cstatus;
       char *gargv[] =
       {"Child", "ARGC1", "ARGC2", "Parent", 0};
       if ((status = vfork()) != 0) {
              if (status < -1)
                     printf("Parent - Child process failed\n");
              else {
                     printf("Parent - waiting for Child\n");
                     if ((status = wait(&cstatus)) == -1)
                            perror("Parent - Wait failed");
                     else if (cstatus == CLI$_IMAGEFNF)
                            printf("Parent - Child does not exist\n");
                     else
                            printf("Parent - Child final status: %x\n",
                                   cstatus);
              }
       }
       else {
              printf("Parent - Starting Child\n");
              if ((status = execv(child_name, gargv)) == -1) {
                     perror("Parent - Exec failed");
                     exit(EXIT_FAILURE);
              }
       }
}
-------------------------------------------------------
/* This is a child program that echos its arguments */
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
       int i;
       printf("Program name: %s\n", argv[0]);
       for (i = 1; i < argc; i++)
              printf("Argument %d: %s\n", i, argv[i]);
       return(255) ;
}
```

Checking the Status of Child Processes

```
/* In this example 5 child processes are started. The wait() */
/* function is placed in a separate for loop so that it is */
/* called once for each child. If wait() were called within */
/* the first for loop, the parent would wait for one child to */
/* terminate before executing the next child. If there were */
/* only one wait request, any child still running when the */
/* parent exits would terminate prematurely. */
#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>
const char *child_name = "check_stat_child.exe" ;
main()
{
       int status,
       cstatus,
       i;
       for (i = 0; i < 5; i++) {
              if ((status = vfork()) == 0) {
                     printf("Parent - Starting Child %d\n", i);
                     if ((status = execl(child_name, 0)) == -1) {
                            perror("Parent - Exec failed");
                            exit(EXIT_FAILURE);
                     }
              }
              else if (status < -1)
                     printf("Parent - Child process failed\n");
       }
       printf("Parent - Waiting for children\n");

       for (i = 0; i < 5; i++) {
              if ((status = wait(&cstatus)) == -1)
                     perror("Parent - Wait failed");
              else if (cstatus == CLI$_IMAGEFNF)
                     printf("Parent - Child does not exist\n");
              else
                     printf("Parent - Child %X final status: %d\n",
                            status, cstatus);
       }
}
```

Communicating Through a Pipe

```
/* In this example, the parent writes a string to the pipe for */
/* the child to read. The child then writes the string back */
/* to the pipe for the parent to read. The wait function is */
/* called before the parent reads the string that the child has */
/* passed back through the pipe. Otherwise, the reads and */
/* writes will not be synchronized. */
#include <perror.h>
#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <processes.h>
#include <unixio.h>
#define inpipe 11
#define outpipe 12
const char *child_name = "pipe_child.exe" ;
main()
{
       int pipes[2];
       int mode, status, cstatus, len;
```

```
        char *outbuf, *inbuf;
        if ((outbuf = malloc(512)) == 0) {
                printf("Parent - Outbuf allocation failed\n");
                exit(EXIT_FAILURE);
        }
        if ((inbuf = malloc(512)) == 0) {
                printf("Parent - Inbuf allocation failed\n");
                exit(EXIT_FAILURE);
        }
        if (pipe(pipes) == -1) {
                printf("Parent - Pipe allocation failed\n");
                exit(EXIT_FAILURE);
        }
        dup2(pipes[0], inpipe);
        dup2(pipes[1], outpipe);
        strcpy(outbuf, "This is a test of two-way pipes.\n");
        status = vfork();
        switch (status) {
        case 0:
                printf("Parent - Starting child\n");
                if ((status = execl(child_name, 0)) == -1) {
                        printf("Parent - Exec failed");
                        exit(EXIT_FAILURE);
                }
                break;
        case -1:
                printf("Parent - Child process failed\n");
                break;
        default:
                printf("Parent - Writing to child\n");
                if (write(outpipe, outbuf, strlen(outbuf) + 1) == -1) {
                        perror("Parent - Write failed");
                        exit(EXIT_FAILURE);
                }
                else {
                        if ((status = wait(&cstatus)) == -1)
                                perror("Parent - Wait failed");
                        if (cstatus == CLI$_IMAGEFNF)
                                printf("Parent - Child does not exist\n");
                        else {
                                printf("Parent - Reading from child\n");
                                if ((len = read(inpipe, inbuf, 512)) <= 0) {
                                        perror("Parent - Read failed");
                                        exit(EXIT_FAILURE);
                                }
                                else {
                                        printf("Parent: %s\n", inbuf);
                                        printf("Parent - Child final status: %d\n",
                                        cstatus);
                                }
                        }
                }
                break;
        }
}
-----------------------------------------------------------------------
/* This is a child program which reads from a pipe and writes */
/* the received message back to its parent. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define inpipe 11
#define outpipe 12
main()
{
        char *buffer;
        int len;
        if ((buffer = malloc(512)) == 0) {
```

```
            perror("Child - Buffer allocation failed\n");
            exit(EXIT_FAILURE);
      }
      printf("Child - Reading from parent\n");
      if ((len = read(inpipe, buffer, 512)) <= 0) {
            perror("Child - Read failed");
            exit(EXIT_FAILURE);
      }
      else {
            printf("Child: %s\n", buffer);
            printf("Child - Writing to parent\n");
            if (write(outpipe, buffer, strlen(buffer) + 1) == -1) {
                  perror("Child - Write failed");
                  exit(EXIT_FAILURE);
            }
      }
      exit(EXIT_SUCCESS);
}
```

4.7.2. Communication Between Parent and Child Process:

On UNIX, the parent process communicates with the child process using the pipes established by the parent. Once fork, the child process redirects its standard channels (stdin, stdout and stderr) to the pipes while the parent process watches the pipes for child's stdout and stderr once the child is spawned by the call to exec.

The UNIX's scheme is as follows:

```
parent:
     create pipes for stdin, stdout, stderr
     fork
child:
     remap(0 to the pipe for stdin);
     remap(1 to the pipe for stdout);
     remap(2 to the pipe for stderr);
     exec
     exit
parent:
     watch the pipes for child's stdout and stderr
```

On OpenVMS, vfork() does not create a new child until the exec() routine is called, so the sontext is still with parent process. Therefore, to communicate through the pipes, the parent and the child must agree about the numbers of file descriptor which will be associated with the pipes. The parent must redirect the pipes to the predefined file descriptors before the call to exec() while the child must redirect its standard channels to the predefined file descriptors by itself.

The routine, decc$set_child_standard_streams() allows the user to tell the C RTL to associate certain files with child' standard channels before the exec() request is issued.

OpenVMS's scheme is as follows:

```
parent:
     /* create pipes for stdin, stdout, stderr */
     int fdin[2], fdout[2], fderr[2];
     pipe(fdin);
     pipe(fdout);
     pipe(fderr);
     /* call decc$set_child_standard_streams() */
     decc$set_child_standard_streams(fdin[0], fdout[1], fderr[1]);
```

```
    /* spawn the child */
     vfork
     exec
child:
     read from stdin, write to stdout/stderr
     exit
parent:
     write to fdin[1], read from fdout[0] and fderr[0] for child's
     stdout and stderr
```

The decc$set_child_standard_streams() routine stores specified file descriptors in a thread-specific memory, so that different threads can specify different mapping for child' standard channels.

4.7.3. Handling Client Server Applications

One of common issues while porting applications, especially client or server applications from UNIX or Linux to OpenVMS is because of the fork() call. This is typically handled on UNIX using "fork" or "inetd". The newly created process handles the client connection, whereas the original process waits in infinite loop for the new client connection.

On OpenVMS, one of the commonly used mechanisms is to use TCP Auxiliary Server concept to accept the client connection on a particular port. Application server must use socket API and use the TCPIP$C_AUXS option to accept this connection as follows:

```
socket (TCPIP$C_AUXS, 0, 0)
```

Where; TCPIP$C_AUXS is defined as 127

TCPIP$C_AUXS option accept and hand-off of a socket is already created and initialized by the auxiliary server.

The following procedure has been adopted in Samba:

Set the service in TCP/IP using the well known port as follows:

```
$ tcpip set service smbd /protocol=TCP /port=139 /flag=listen /limit=100
/user=SAMBA$SMBD /process=SMBD /file=SMBD_STARTUP.COM
/log=(FILE:SMBD_STARTUP.LOG,ALL)
```

Where;
"/file" indicates the Name of the startup command file

"/port" indicates the service to be used.

"/user" the username with which the new client connections are handled.

Whenever there is any activity in the above mentioned port the command procedure mentioned in the service gets executed, which has call socket(127,0,0). This socket returns the socket descriptor, which can be used to communicate with the client.

In the command procedure, run the image that has the server code to handle the client connection. A new process gets created which has the name mentioned in "/process" option, in the user "/user".

Set the TCP/IP service and the port that has to accept the new client connection. For example,

```
$ tcpip set service hello -
/port=12345 -
/protocol=tcp -
/user=aravinda -
/process_name=hello_world -
/file=DKA0:[EXAMPLES.PORTING_TIPS.TCPIP]hello_run.com

Create a service run command procedure, named HELLO_RUN.COM that contains the
following lines:

$ define sys$output DKA0:[EXAMPLES.PORTING_TIPS.TCPIP]hello_service.log
$ define sys$error DKA0:[EXAMPLES.PORTING_TIPS.TCPIP]hello_service.log
$ run DKA0:[EXAMPLES.PORTING_TIPS.TCPIP]tcpip$tcp_server_sock_auxs.exe
Run the client program
$ run tcpip$tcp_client_sock
Enter remote host:
```

The auxiliary server receives the connect service request from the client. It creates a process and executes the commands in hello_run.com to run this server program.

This server program then logs client connection information and client data to the service log before replying to the client host with the message - "`Hello, world! Sending info to client...`".

The SERVER and the CLIENT Program that are used are:
TCP/IP Server Program

```
/*
*
* ABSTRACT:
*
* This is an example of a TCP/IP IPv4 server using 4.x BSD
* socket Application Programming Interface (API) to handle
* network I/O operations. In addition, it shows how to
* accept connections from the auxiliary server.
*
*
*/
/* Build, Configuration, and Run Instructions */
/*
* BUILD INSTRUCTIONS:
*
* To build this example program use commands of the form,
*
* using the "C" compiler:
*
* $ cc/prefix=all TCPIP$TCP_SERVER_SOCK_AUXS.C
* $ link TCPIP$TCP_SERVER_SOCK_AUXS
*
*
* CONFIGURATION INSTRUCTIONS:
*
* To configure this example program:
*
* 1) Create a service run command procedure, named HELLO_RUN.COM, that
* contains the following lines:
```

```
*
* $ define sys$output ddcu:[directory]hello_service.log
* $ define sys$error ddcu:[directory]hello_service.log
* $ run ddcu:[directory]tcpip$tcp_server_sock_auxs.exe
*
* where: ddcu:[directory] is the device and directory of where the
* hello service run command procedure file resides
*
* 2) Create a service database entry for the hello service as shown below:
*
* $ tcpip set service hello -
* _$ /port=12345 -
* _$ /protocol=tcp -
* _$ /user=vms_user_account -
* _$ /process_name=hello_world -
* _$ /file=ddcu:[directory]hello_run.com
*
* 3) Enable the hello service to run as shown below:
*
* $ tcpip enable service hello
*
*
* RUN INSTRUCTIONS:
*
* To run this example program:
*
* 1) Start the client program, entering the server host as shown below:
*
* $ run tcpip$tcp_client_sock
* Enter remote host:
*
* Note: You can specify a server host by using either an IPv4
* address in dotted-decimal notation (e.g. 16.20.10.56)
* or a host domain name (e.g. serverhost.hp.com).
*
* 2) The auxiliary server receives the hello service request, creates a
* process, then executes the commands in hello_run.com to run this
* server program. This server program then logs client connection
* information and client data to the service log before replying to
* the client host with a message of "Hello, world!".
*
*/
/*
* INCLUDE FILES:
*/
#include <in.h> /* define internet related constants, */
/* functions, and structures */
#include <inet.h> /* define network address info */
#include <socket.h> /* define BSD 4.x socket api */
#include <stdio.h> /* define standard i/o functions */
#include <stdlib.h> /* define standard library functions */
#include <string.h> /* define string handling functions */
#include <tcpip$inetdef.h> /* define tcp/ip network constants, */
/* structures, and functions */
#include <unixio.h> /* define unix i/o */

/*
* FORWARD REFERENCES:
*/
int main( void ); /* server main
/* Server Main */
/*
* FUNCTIONAL DESCRIPTION:
*
* This is the server's main-line code. It handles all the tasks of the
* server including: socket creation, writing client connection data,
* and terminating client connections.
*
* This example program implements a typical TCP IPv4 server using the
```

```
* BSD socket API to handle network i/o operations. In addition, it
* uses the auxiliary server to accept client connections.
*
* 1) To create a socket:
*
* socket()
*
* 2) To transfer data:
*
* send()
*
* 3) To close a socket:
*
* close()
*
* This function is invoked by the DCL "RUN" command (see below); the
* function's completion status is interpreted by DCL and if needed,
* an error message is displayed.
*
* SYNOPSIS:
*
* int main( void )
*
* FORMAL PARAMETERS:
*
* ** None **
*
* IMPLICIT INPUTS:
*
* ** None **
*
* IMPLICIT OUTPUTS:
*
* ** None **
*
* FUNCTION VALUE:
*
* completion status
*
* SIDE EFFECTS:
*
* ** None **
*
*/
int
main( void )
{
       int sockfd; /* socket descriptor */
       unsigned int client_addrlen; /* returned length of client socket */
       /* address structure */
struct sockaddr_in client_addr; /* client socket address structure */

       /* server data buffer */
       char buf[] = "Hello, world! Sending info to client...";

       /*
       * init client's socket address structure
       */
       memset( &client_addr, 0, sizeof(client_addr) );

       /*
       * create socket
       */
       if ( (sockfd = socket(TCPIP$C_AUXS, SOCK_STREAM, 0)) < 0 )
       {
              perror( "Failed to create socket" );
              exit( EXIT_FAILURE );
       }
       /*
```

```
        * log this client connection
        */
        client_addrlen = sizeof(client_addr);
        if ( getpeername(sockfd,
        (struct sockaddr *) &client_addr, &client_addrlen) < 0 )
        {
                perror( "Failed to accept client connection" );
                exit( EXIT_FAILURE );
        }
        printf( "Accepted connection from host: %s, port: %d\n",
                inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port)
                );
        /*
        * connection established with a client;
        * now attempt to write on this connection
        */
        if ( send(sockfd, buf, sizeof(buf), 0) < 0 )
        {
                perror( "Failed to write data to client connection" );
                exit( EXIT_FAILURE );
        }
        printf( "Data sent: %s\n", buf ); /* output server's data buffer */
        /*
        * close socket
        */
        if ( close(sockfd) < 0 )
        {
                perror( "Failed to close socket" );
                exit( EXIT_FAILURE );
        }
        exit( EXIT_SUCCESS );
}
```

TCPIP Client Program

```
/*
* ABSTRACT:
*
* This is an example of a TCP/IP IPv4 client using 4.x BSD
* socket Application Programming Interface (API) to handle
* network I/O operations.
*
* Refer to 'Build, Configuration, and Run Instructions' for
* details on how to build, configure, and run this program.
*/
/* Build, Configuration, and Run Instructions */
/*
* BUILD INSTRUCTIONS:
*
* To build this example program use commands of the form,
*
* using the "C" compiler:
*
* $ cc/prefix=all TCPIP$TCP_CLIENT_SOCK.C
* $ link TCPIP$TCP_CLIENT_SOCK
* RUN INSTRUCTIONS:
*
* To run this example program:
*
* 1) Start the client's server program as shown below:
*
* $ run tcpip$tcp_server_sock
* Waiting for a client connection on port: m
*
* 2) After the server program blocks, start this client program,
* entering the server host as shown below:
*
* $ run tcpip$tcp_client_sock
```

```
* Enter remote host:
*
* Note: You can specify a server host by using either an IPv4
* address in dotted-decimal notation (e.g. 16.20.10.56)
* or a host domain name (e.g. serverhost.hp.com).
*
* 3) The client program then displays server connection information
* and server data as shown below:
*
* Initiated connection to host: a.b.c.d, port: n
* Data received: Hello, world!
*
* You can enter "ctrl/z" at any user prompt to terminate program
* execution.
*
*/
/*
* INCLUDE FILES:
*/
#include <in.h> /* define internet related constants, */
/* functions, and structures */
#include <inet.h> /* define network address info */
#include <netdb.h> /* define network database library info */
#include <socket.h> /* define BSD 4.x socket api */
#include <stdio.h> /* define standard i/o functions */
#include <stdlib.h> /* define standard library functions */
#include <string.h> /* define string handling functions */
#include <unixio.h> /* define unix i/o */
/*
* NAMED CONSTANTS:
*/
#define BUFSZ 1024 /* user input buffer size */
#define SERV_PORTNUM 12345 /* server port number */
/*
* FORWARD REFERENCES:
*/
int main( void ); /* client main */
void get_serv_addr( void * ); /* get server host address */
/* Client Main */
/*
* FUNCTIONAL DESCRIPTION:
*
* This is the client's main-line code. It handles all the tasks of the
* client including: socket creation, initiating server connections,
* reading server connection data, and terminating server connections.
*
* This example program implements a typical TCP IPv4 client using the
* BSD socket API to handle network i/o operations as shown below:
*
* 1) To create a socket:
*
* socket()
*
* 2) To initiate a connection:
*
* connect()
*
* 3) To transfer data:
*
* recv()
*
* 4) To shutdown a socket:
*
* shutdown()
*
* 5) To close a socket:
*
* close()
*
```

```
* This function is invoked by the DCL "RUN" command (see below); the
* function's completion status is interpreted by DCL and if needed,
* an error message is displayed.
*
* SYNOPSIS:
*
* int main( void )
*
*
*/
int
main( void )
{
       int sockfd; /* connection socket descriptor */
       char buf[512]; /* client data buffer */
      struct sockaddr_in serv_addr; /* server socket address structure */
       /*
       * init server's socket address structure
       */
       memset( &serv_addr, 0, sizeof(serv_addr) );
       serv_addr.sin_family = AF_INET;
       serv_addr.sin_port = htons( SERV_PORTNUM );
       get_serv_addr( &serv_addr.sin_addr );
       /*
       * create connection socket
       */
       if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
       {
              perror( "Failed to create socket" );
              exit( EXIT_FAILURE );
       }
       /*
       * connect to specified host and port number
       */
       printf( "Initiated connection to host: %s, port: %d\n",
              inet_ntoa(serv_addr.sin_addr), ntohs(serv_addr.sin_port)
              );
       if ( connect(sockfd,
       (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0 )
       {
              perror( "Failed to connect to server" );
              exit( EXIT_FAILURE );
       }
       /*
       * connection established with a server;
       * now attempt to read on this connection
       */
       if ( recv(sockfd, buf, sizeof(buf), 0) < 0 )
       {
              perror( "Failed to read data from server connection" );
              exit( EXIT_FAILURE );
       }
       printf( "Data received: %s\n", buf ); /* output client's data buffer */
       /*
       * shutdown connection socket (both directions)
       */
       if ( shutdown(sockfd, 2) < 0 )
       {
              perror( "Failed to shutdown server connection" );
              exit( EXIT_FAILURE );
       }
       /*
       * close connection socket
       */
       if ( close(sockfd) < 0 )
       {
              perror( "Failed to close socket" );
              exit( EXIT_FAILURE );
       }
```

```
	exit( EXIT_SUCCESS );
}
/* Get Server Host Address */
/*
* FUNCTIONAL DESCRIPTION:
*
* This function gets the server host's address from the user and then
* stores it in the server's socket address structure. Note that the
* user can specify a server host by using either an IPv4 address in
* dotted-decimal notation (e.g. 16.20.10.126) or a host domain name
* (e.g. serverhost.hp.com).
*
* Enter "ctrl/z" to terminate program execution.
*
* SYNOPSIS:
*
* void get_serv_addr( void *addrptr )
*
* FORMAL PARAMETERS:
*
* addrptr - pointer to socket address structure's 'sin_addr' field
* to store the specified network address
*
* IMPLICIT INPUTS:
*
* ** None **
*
* IMPLICIT OUTPUTS:
*
* ** None **
*
* FUNCTION VALUE:
*
* ** None **
*
* SIDE EFFECTS:
*
* Program execution is terminated if unable to read user's input
*
*/
void
get_serv_addr( void *addrptr )
{
	char buf[BUFSZ];
	struct in_addr val;
	struct hostent *host;
	while ( TRUE )
	{
		printf( "Enter remote host: " );
		if ( fgets(buf, sizeof(buf), stdin) == NULL )
		{
			printf( "Failed to read User input\n" );
			exit( EXIT_FAILURE );
		}
		buf[strlen(buf)-1] = 0;
		val.s_addr = inet_addr( buf );
		if ( val.s_addr != INADDR_NONE )
		{
			memcpy( addrptr, &val, sizeof(struct in_addr) );
			break;
		}
		if ( (host = gethostbyname(buf)) )
		{
			memcpy( addrptr, host->h_addr, sizeof(struct in_addr) );
			break;
		}
	}
}
```

4.7.4. Writing a Daemon Server Process to Handle the new Client Connection

Generally on Linux, when a new child process starts, the descriptors (file, socket, terminal and so on) that are open gets shared between parent and child. This is useful for writing Daemon servers in which child processes handles the new client connection and parent handles the new incoming client connection. But on an OpenVMS system this is not possible because the fork is not available and descriptors are not shared between the processes.

If you write a Daemon Server process that waits for a request, and whenever a new client request arrives, a new process gets created. Then you will have to hand off the connection to the new process.

You cannot pass the descriptor number to the newly created process as this number has no significance in the new process. Convert this descriptor number to the device ID as the sockets are basically devices in OpenVMS and pass this device ID to a newly created process and reconvert it back to the descriptor number so that you can use various C RTL and socket routines on this descriptor.
To convert the socket descriptor to device IO channel use the following socket API function:

```
decc$get_sdc()
```

This API returns the socket device's OpenVMS I/O channel Socket Device Channel (SDC) associated with a socket descriptor.

Format

```
#include <socket.h>
short int decc$get_sdc ( int s );
```

To convert the device IO channel to socket descriptor, use the following Socket API:

```
decc$socket_fd
```

This API returns the socket descriptor associated with a SDC

Format
```
#include <socket.h>
int decc$socket_fd (int channel);
```

This method has been used in the Advanced Server for VMS (ASV) product to accept the client's connection in one OpenVMS process (PWRK$LMMCP) and then hands off that connection to a different process (PWRK$LMSRV). The process that gets the connection handed off processes the client's further connection.

4.8. User or Group Issues

4.8.1. User “root” Related Issues

The “root” user has a special significance on the UNIX or LINUX systems. Many applications on UNIX or Linux systems make this user equivalent to the SYSTEM user. This is not applicable to OpenVMS because any user who has the privileges can act as the “root” user. Using the “system” account as an equivalent to “root” also causes problems during “audit”.

To correct this problem, designate one of the accounts to act as “root” in your application. For example, “samba$smbd” is “root” in samba. In most cases, UNIX or LINUX applications assume (setuid) that they are working on “root” or UID of 0. Write wrapper routines for the setuid calls to address this issue.
OpenVMS provides various UNIX compliant functions, such as setuid, getuid, getgid, setgid and so on by enabling POSIX style Ids. POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.
However, these calls must be modified on OpenVMS systems for the following reasons:

- POSIX IDs are global and hence these ids affect the existing applications on the system. This is not advised.
- Causes unknown fallouts on other programs that are functioning on the system.
- Enabling of POSIX IDs is not documented or supported.

NOTE: To enable POSIX IDs define the DECC$POSIX_STYLE_UID logical to "enable"
You can write wrapper routines using the PERSONA system services namely, $GET_PERSONA, $SET_PERSONA .

4.8.2. UNIX Group Related Issues

The Group concept available in OpenVMS is not equivalent to the UNIX group and is not sufficient to handle the UNIX application’s requirements.

The authorize database needs to be modified for any group related management on OpenVMS viz. adding an user to the group, deleting an user from a group, listing all the users of the group, getting group name and so on. This is not acceptable in many cases for handling UNIX applications on OpenVMS.

Also, all the group related C RTL APIs must work as in UNIX. setgid() function must return a particular value for a particular Group. And getgid() must be handled to return values that are set previously by setgid() functions.

Getgrgid: When 'gid' is provided, getgrgid fetches the struct group information
Getgrnam: When the group name is provided, getgrnam fetches the group information
Getgrouplist: Fetches the list of IDs held by 'user'. Also, includes 'group' in the list.
Setgroups: Grants a list of identifiers to the PROCESS
Getgroups: Fetches the list (and/or count) of the IDs held by the PROCESS
Setgid: Sets the default ID for the user/process.
Getgid: Fetches the default ID of the user/process.
Initgroups: Fetches the list of identifiers held by the 'user' and includes 'gid' to the list. It grants the list of IDs to the process.

To address this problem simulate the “group” concept using OpenVMS identifiers as follows:

1. Add Resource IDENTIFIERs in the Authorize database:

```
UAF> ADD/IDENTIFIER/ATTRIBUTES=(RESOURCE) ADMIN
UAF> show/identifier ADMIN
      Name                            Value           Attributes
      ADMIN                           %X88001211      RESOURCE
```

2. To make the user a member of the group, grant the IDENTIFIER to the user.

```
UAF> grant/identifier ADMIN TEST1
UAF> grant/identifier ADMIN TEST3
UAF> grant/identifier ADMIN TEST3
```

3. Set the restriction on the object using the added Identifier.

```
$ set security test.txt /ACL=(IDENTIFIER=ADMIN,ACCESS=READ) /DELETE=ALL
$ dir test.txt/sec
     Directory DKA0:[EXAMPLES]
      TEST.TXT;1           [TEST1]
(IDENTIFIER=ADMIN,ACCESS=READ)
```

Only users holding the identifier ADMIN has the privilege to READ the "test.txt" file.

Currently this method is adopted in Samba on OpenVMS.

4.8.3. Windows ACLs

Most of the Windows based applications have Access Control List (ACL) on an object for security. If the server is based on the open source applications such as Samba, it is expected that all the objects that the client (windows) applications deals with will have ACL on the object indicating the security. ACL provides fine tuned access to a particular object namely, Deny, which allows a particular member in the group.

A set of ACL APIs are available on UNIX/Linux to manage Windows style access control. OpenVMS does not provide an API for ACL handling. A few of the APIs provided on UNIX or Linux are:

acl_init	initializes ACL working storage
acl_add_perm	adds permission to an ACL permission set
acl_check	checks an ACL for validity
acl_clear_perms	clears all permissions from an ACL permission set
acl_set_file	sets an ACL by filename
acl_get_file	gets an ACL by filename
acl_get_perm	tests for a permission in an ACL permission set

The following system services can be used on OpenVMS to write the equivalent ACL APIs.

```
sys$set_security
sys$get_security
sys$parse_acl
sys$idtoasc
```

4.9. Terminal Handling Example

The following example illustrates how to read information from the terminals:

```
#include <stdlib.h>
#include <string.h>

#if defined(__VMS)
#define __NEW_STARLET
#include <lib$routines.h>   /* LIB$ RTL-routine signatures.*/
#include <starlet.h>        /* Sys ser calls */
```

```
#if defined(__VAX) && __VMS_VER /* Don't check vms versions.  iledef.h not
available on VAX < 70200000 */
typedef struct _ile3 {
    unsigned short int ile3$w_length;  /* Length of buffer in bytes        */
    unsigned short int ile3$w_code;    /* Item code value                  */
    void *ile3$ps_bufaddr;             /* Buffer address                   */
    unsigned short int *ile3$ps_retlen_addr; /*Address of word for returned
length */
    } ILE3;

typedef struct _iosb {
    unsigned short int iosb$w_status;   /* Final I/O status             */
    unsigned short int iosb$w_bcnt;     /* 16-bit byte count            */
    unsigned int iosb$l_dev_depend;     /* 32-bit device dependent info */
    } IOSB;

#define TTY$C_CTRLZ 26
#else
#include <iledef.h>
#include <iosbdef.h>
#include <ttysymdef.h>
#endif

#include <ssdef.h>
#include <dcdef.h>
#include <dvidef.h>
#include <ttdef.h>
#include <iodef.h>
#include <descrip.h>

#undef  __NEW_STARLET

/*
** Literals - used in prompter.c and also defined there.
** Make sure they stay in synch.
*/
#define ECHO_ON  1
#define ECHO_OFF 0
#define TRUE  1
#define FALSE 0

/*
** Prototypes
*/
int tt_open
    (
    volatile unsigned int *tt_chan
    );

int tt_read
    (
    unsigned int tt_chan,
    char *input_buf,
    int input_max,
    int read_echo
    );

int tt_close
    (
    unsigned int tt_chan
    );

unsigned int dev_class;

#endif /* __VMS */

#if defined(__VMS)
/*
```

```
**
**  tt_open - This routine opens the channel to the terminal
**
**  Functional Description:
**
**      This routine opens the channel to the terminal.
**
**  Usage:
**
**      tt_open tt_chan
**
**  Formal parameters:
**
**      tt_chan    Out    Address to receive the terminal channel number
**
**  Implicit Parameters:
**
**      None
**
**  Routine Value:
**
**      None
**
**  Side Effects:
**
**      None
**
*/
int
tt_open
    (
    volatile unsigned int  *tt_chan
    )
{
    ILE3 dvi_list[2] = {0,0,0,0,0,0,0,0};
    int status;
    IOSB iosb;
    $DESCRIPTOR(tt_desc, "SYS$COMMAND");

    /*
    ** Setup GETDVI item list
    */
    dvi_list[0].ile3$w_length = sizeof (dev_class);
    dvi_list[0].ile3$w_code = DVI$_DEVCLASS;
    dvi_list[0].ile3$ps_bufaddr = &dev_class;

    /*
    ** Get device type for SYS$COMMAND
    */
    status = sys$getdviw (0, 0,
                                       &tt_desc,
                                       &dvi_list,
                                       &iosb,
                                       0, 0, 0);
    if (status & 1)
        status = iosb.iosb$w_status;
    if (status != SS$_NORMAL)
        exit (status);

    /*
    ** Assign channel to terminal
    */
    status = sys$assign (&tt_desc,
                         (unsigned short*) tt_chan,
                         0, 0);
    if (! (status & 1))
        exit (status);

    /*
```

```
    ** Return success
    */
    return (TRUE);
}

/*
**
**  tt_read - This routine read from the terminal
**
**  Functional Description:
**
**      This routine reads from the terminal.
**
**  Usage:
**
**      tt_close tt_chan
**
**  Formal parameters:
**
**      tt_chan   In    The channel number of the terminal
**
**  Implicit Parameters:
**
**      None
**
**  Routine Value:
**
**      None
**
**  Side Effects:
**
**      None
**
*/
int
tt_read
    (
    unsigned int tt_chan,
    char *input_buf,
    int input_max,
    int read_echo
    )
{
    struct dsc$descriptor_s InputDesc = {0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0};
    unsigned short InputLen;
    int read_func,
    status;
    IOSB iosb;
    #if __INITIAL_POINTER_SIZE == 64
    #pragma __required_pointer_size __save
    #pragma __required_pointer_size 32
    #endif
    char *TmpInputBuf;

    /* If input_max comes in as zero,
     * we can not allow a read on the terminal,
     * nor can we allocate a pointer of zero bytes.
     * Just give up, the caller needs to know what
     * to do.  Set the reply length to zero.
     */
    if(input_max <= 0)
    {
        input_buf[0] = '\0';
        return(TRUE);
    }

   #if !defined(__VAX) && __INITIAL_POINTER_SIZE
      TmpInputBuf = (char*) _malloc32 (input_max);
   #else
```

```
      TmpInputBuf = (char*) malloc (input_max);
   #endif

   #if __INITIAL_POINTER_SIZE == 64
     #pragma __required_pointer_size __restore
   #endif

    /*memset (TmpInputBuf, '\0', input_max);*/
    InputLen = 0;

    /*
    ** Determine if the input device is a terminal ?
    */
    if (dev_class == DC$_TERM)
    {
        /*
        ** Set the read function based on the desired echo
        */
        if (read_echo == ECHO_OFF)
            read_func = IO$_READVBLK | IO$M_NOECHO;
        else
            read_func = IO$_READVBLK;

        /*
        ** Read from the terminal
        */
        status = sys$qiow (0,
                           tt_chan,
                           read_func,
                           &iosb,
                           0, 0,
                           TmpInputBuf,
                           input_max - 1,
                           0, 0, 0, 0);
        if (status != SS$_NORMAL && !(iosb.iosb$w_status & 1))
            InputLen = 0;
        else
            InputLen = iosb.iosb$w_bcnt;
    }
    else
    {
        InputDesc.dsc$w_length = input_max - 1;
        InputDesc.dsc$a_pointer = TmpInputBuf;
        status = lib$get_command (&InputDesc, 0, &InputLen);
        if (status != SS$_NORMAL)
            exit (status);
    }
    /*
    ** Check to see if Control-Z was entered.  Copy if not.
    */

    TmpInputBuf[InputLen] = '\0';
    if (! strchr (TmpInputBuf, TTY$C_CTRLZ)) {
        strcpy (input_buf, TmpInputBuf);
    };
    /*
    ** Free the temporary Input buffer
    */
    free (TmpInputBuf);

    /*
    ** Return success
    */
    return (TRUE);
}


/*
**
```

```
**  tt_close - This routine closes the channel to the terminal
**
**  Functional Description:
**
**      This routine closes the channel to the terminal.
**
**  Usage:
**
**      tt_close tt_chan
**
**  Formal parameters:
**
**      tt_chan  In  The channel number of the terminal
**
**  Implicit Parameters:
**
**      None
**
**  Routine Value:
**
**      None
**
**  Side Effects:
**
**      None
**
*/
int
tt_close
    (
    unsigned int tt_chan
    )
{
    int status;

    /*
    ** Deassign channel to terminal
    */
    status = sys$dassgn (tt_chan);
    if (! (status & 1))
        exit (status);

    /*
    ** Return success
    */
    return (TRUE);
}
#endif /* __VMS */
```

4.10. mmap() limitations

mmap() opens the files in a binary mode and you can open the file if the file is in stream –lf or in binary format. However, other types of RMS file fail to open when the program encounters the embedded RMS information. Mmap on OpenVMS requires input files to be either stream-lf or binary.

4.11. TCP/IP

4.11.1. UNIX Domain Sockets

OpenVMS does not support the domain sockets feature. However, equivalent functionality can be provided using the existing OpenVMS IPC mechanisms. Since this needs huge source code changes, an alternative approach is described in this section.

Use the local host 127.0.0.1 loop back address to communicate using a predefined and a known port. Also, differentiate the normal socket descriptor and the socket descriptor generated using this mechanism, as sockets are treated as files on UNIX but devices on VMS.

4.11.2. Select() call on file descriptors

Select() call is extensively used in UNIX variants for asynchronous I/O and to handle I/O from multiple descriptors at the same time. However in OpenVMS Select() call does not manage the file descriptors and manages only the socket descriptors for remote communication. You can handle this limitation in OpenVMS using the QIO and AST mechanism.

The following program illustrates what error is returned when select call is called on file descriptors:

```
#include <socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <types.h>
#include <unistd.h>
int  main(void)
{
        fd_set rfds;
        struct timeval tv;
        int retval;
        /* Watch stdin (fd 0) to see when it has input. */
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        /* Wait up to five seconds. */
        tv.tv_sec = 5;
        tv.tv_usec = 0;
        retval = select(1, &rfds, NULL, NULL, &tv);
        /* Don't rely on the value of tv now! */
        if (retval == -1)
                perror("select()");
        else if (retval)
               printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
        else
                printf("No data within five seconds.\n");
        return 0;
}
```

4.12. Pthreads- sleep() routine thread safety

The C RTL sleep routine may hang if upcalls are enabled for the process. The hang does not occur when upcalls are disabled. In such a case, it is recommended to use pthread_cond_timedwait() for sleep implementation.

The following is a pseudo code implementation of a sleep routine using pthread_cond_timedwait:

```
Sleep2(unsigned seconds)
{
    pthread_cond_init (&cond, NULL);
```

```
    pthread_mutex_init (&mutx, NULL);
    pthread_get_expiration_np (&deltaTime, &absTime);

    pthread_cond_timedwait (&cond, &mutx, &absTime);

    pthread_mutex_unlock (&mutx);

    pthread_cond_destroy (&cond);
    pthread_mutex_destroy (&mutx);
}
```

4.13. ioctl for terminals

If you encounter ioctl() calls being used to control a terminal, rewrite the code using QIO's. You can access the documentation from the source UNIX, as ioctl is not portable amongst UNIX systems.

4.13.1. Example of ioctl Implementation for OpenVMS

An implementation of ioctl on OpenVMS is as follows:

```
/*
** VMS$IOCTL INCLUDE FILES
*/
#include <descrip.h>        /* VMS descriptor stuff */
#include <errno.h>          /* UNIX style error codes for IO routines. */
#include <iodef.h>          /* I/O FUNCTION CODE DEFS */
#include <lib$routines.h>   /* LIB$ RTL-routine signatures.*/
#include <ssdef.h>          /* SS$_<xyz> sys ser return stati <8-)
*/

#include <starlet.h>        /* Sys ser calls */
#include <stdio.h>          /* UNIX 'Standard I/O' Definitions   */
#include <stdlib.h>         /* General Utilities */
#include <string.h>         /* String handling function definitions */
#include <ucx$inetdef.h>    /* UCX network definitions */
#include <socket.h>         /* TCP/IP socket definitions. */

/*
** VMS$IOCTL MACRO DEFINITIONS
*/
#ifndef _IO
#define IOCPARM_MASK    0x7f            /* Parameters are < 128 bytes   */
#define IOC_VOID        (int)0x20000000 /* No parameters                */
#define IOC_OUT         (int)0x40000000 /* Copy out parameters          */
#define IOC_IN          (int)0x80000000/* Copy in parameters           */
#define IOC_INOUT       (int)(IOC_IN|IOC_OUT)
#define _IO(x,y)        (int)(IOC_VOID|('x'<<8)|y)
#define _IOR(x,y,t)     (int)(IOC_OUT|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)|y)
#define _IOW(x,y,t)     (int)(IOC_IN|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)|y)
#define _IOWR(x,y,t)    (int)(IOC_INOUT|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)`
#endif
#define VMSOK(s) (s & 01)

#if __INITIAL_POINTER_SIZE == 64
#pragma __required_pointer_size __save
#pragma __required_pointer_size 32
#endif
int vms$ioctl(int d, int request, char *argp);
#if __INITIAL_POINTER_SIZE == 64
#pragma __required_pointer_size __restore
#endif

/*
** Functional Description
```

```
**
** Ioctl's have the command encoded in the lower word,
** and the size of any in or out parameters in the upper
** word.  The high 2 bits of the upper word are used
** to encode the in/out status of the parameter; for now
** we restrict parameters to at most 128 bytes.
** The IOC_VOID field of 0x20000000 is defined so that new ioctls
** can be distinguished from old ioctls.
**
** Formal Parameters
**           d...file or socket descriptor
**           request...defined in ioctl.h
**           argp..."in" or "out" parameter
**
** Routine Value
**
**           Status code
*/
#if __INITIAL_POINTER_SIZE == 64
#pragma __required_pointer_size __save
#pragma __required_pointer_size 32
#endif
int vms$ioctl (
    int d,
    int request,
    char *argp
    )
{
int ef;                                 /* Event flag number */
int sdc;                                /* Socket device channel */
unsigned short fun;                     /* Qiow function code  */
unsigned short iosb[4];                 /* Io status block */
char *p5, *p6;                          /* Args p5 & p6 of qiow */
struct comm {
    int command;
    char *addr;
    } ioctl_comm;                       /* Qiow ioctl commands. */
struct it2 {
    unsigned short len;
    unsigned short opt;
    struct comm *addr;
    } ioctl_desc;                       /* Qiow ioctl commands descriptor */
int status;

/*
** Gets an event flag for qio
*/
status = lib$get_ef (&ef);
if (!VMSOK(status))
    {
    /*
    ** No ef available. Use 0
    */
    ef = 0;
    }

/*
** Get the socket device channel number.
*/
sdc = vaxc$get_sdc (d);
if (sdc == 0)
    {
    /*
    ** Not an open socket descriptor.
    */
#ifdef DEBUG
    printf ("vaxc$get_sdc failed\n");
#endif
    errno = EBADF;
```

```
    status = lib$free_ef (&ef);
    return -1;
    }

/*
** Fill in ioctl descriptor.
*/
ioctl_desc.opt = INET$C_IOCTL;
ioctl_desc.len = sizeof (struct comm);
ioctl_desc.addr = &ioctl_comm;

/*
** Decide qio function code and in/out parameter.
*/
if (request & IOC_OUT)
    {
    fun = IO$_SENSEMODE;
    p5 = 0;
    p6 = (char *) &ioctl_desc;
    }
else
    {
    fun = IO$_SETMODE;
    p5 = (char *) &ioctl_desc;
    p6 = 0;
    }

/*
** Fill in ioctl command.
*/
ioctl_comm.command = request;
ioctl_comm.addr = argp;

/*
** Do ioctl.
*/
status = sys$qiow (ef, sdc, fun, iosb, 0, 0,
                   0, 0, 0,0,             /* p1 - p4: not used */
                   p5, p6);
if (! VMSOK(status))
    {
#ifdef DEBUG
    printf ("ioctl failed: status = %08x\n", status);
#endif
    errno = status;
    status = lib$free_ef (&ef);
    return -1;
    }

if (! VMSOK(iosb[0]))
{
#ifdef DEBUG
printf ("ioctl failed: status = %x, %x, %x%x\n", iosb[0], iosb[1], iosb[3],
iosb[2]);
#endif
    errno = iosb[0];
    status = lib$free_ef (&ef);
return -1;

}

status = lib$free_ef (&ef);

return 0;
}
#if __INITIAL_POINTER_SIZE == 64
#pragma __required_pointer_size __restore
#endif
```

5. For more information

For more information, see the HP C Run-Time Library Reference Manual for OpenVMS Systems.

6. Feedback or Suggestion

If you have experience in porting applications from UNIX or Linux to OpenVMS and want to contribute to this document or if you have any suggestions to make this document better and complete, please send a mail with your content or suggestions to: UNIXtoVMSporting@hp.com.

May 2010