UNIX Code Migration Guide

UNIX Application Migration Guide

Chapter 10: Interix Code Conversion

Larry Twork, Larry Mead, Bill Howison, JD Hicks, Lew Brodnax, Jim McMicking, Raju Sakthivel, David Holder, Jon Collins, Bill Loeffler
Microsoft Corporation

October 2002

Applies to:
Microsoft® Windows®
UNIX applications

The *patterns & practices* team has decided to archive this content to allow us to streamline our latest content offerings on our main site and keep it focused on the newest, most relevant content. However, we will continue to make this content available because it is still of interest to some of our users.
We offer this content as-is, without warranty that it is still technically accurate as some of the material is undoubtedly outdated. Note that the content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Summary: Interix provides a POSIX compatible runtime environment with additional extensions on Windows. This enabled a large number of UNIX applications to run on Windows with little or no adjustment. Chapter 10: Interix Code Conversion discusses in detail the changes you may need to make so that your current application can compile and run on Interix. (71 printed pages)

Contents

Code Examples

Introduction

This chapter describes how to modify UNIX source code to compile on Interix.

Because Interix is designed to be similar to UNIX, few changes are needed to recompile code under Interix. However, there are differences between Interix and UNIX, which mainly arise from the underlying differences in operating system kernels and environments. As a result, some code changes are needed. This chapter discusses how to change those areas that are different.

This chapter first addresses how to convert code. The bulk of this chapter covers the same coding categories as Chapter 9, Win32 Code Conversion. As Chapter 9 describes, many coding changes are needed for Win32. For Interix, only the following categories change and only they are discussed here:

- Processes
- Signals and signal handling
- Threads
- Memory management
- Users, groups and security
- File and data access
- Interprocess communication
- Sockets and networking
- Process environment
- Daemons and services
- Miscellaneous APIs

Within each category, this chapter's sections:

- Describe the coding differences
- Outline options for converting the code
- Illustrate with source code examples

The information presented here is a guide to help developers choose the solution appropriate to an application. Examples are presented as a basis for migrating code.

It is the intention of this guide to give sufficient information to choose the best method for converting the code. After choosing a method, refer to the standard documentation for details of the Microsoft Interix® functions and application programming interfaces (APIs).

How to Convert the Code

To add or isolate code that implements Interix-specific features, surround it with **#ifdef __INTERIX**, such as in the following example:

```
#ifdef __INTERIX
(void) fcntl(fd, F_SETFL, fcntl(fd, F_GETFD) | FD_CLOEXEC));
#else
(void) ioctl(fd, FIOCLEX, NULL);
#endif
```

The **__INTERIX** macro is automatically defined to be true (1) by the c89 compiler interface and by the gcc compiler. The **_POSIX_** macro is defined to be true (1) by c89. The c89 interface also passes the **/Za** option to the Microsoft Visual C++ compiler, which defines **__STDC__**, unless **-N nostdc** is specified. When **-N nostdc** is defined, the ANSI-only mode in the Visual C++ compiler is disabled and Microsoft extensions are allowed. The default is ANSI C mode.

The cc compiler interface defines the symbols **__INTERIX** and **unix** to be true. (The **unix** macro is defined because many applications intended to be compiled on multiple platforms use this macro to call out features found on UNIX systems.) The cc interface also passes the **/Ze** option, which enables language extensions.

The Interix header files are structured to align with the Single UNIX Specification. For example, string and memory functions that occur in POSIX.1 are in string.h. String and memory functions that are in the Single UNIX Specification, but not in POSIX.1, are in strings.h.

The include files are also structured to restrict the API namespace. If the macro **_POSIX_SOURCE** value is defined to be 1 before the first header file is included, the program is restricted to the POSIX namespace. It contains only those APIs specified in the POSIX standards. This can be restrictive.

To get all of the APIs provided with the Interix Software Development Kit (SDK), define the **_ALL_SOURCE** value as 1 before the first header file is included, as shown in the following example:

```
#define _ALL_SOURCE 1
#include <unistd.h>
```

If the source is not defined, the default is the more restrictive **_POSIX_SOURCE**.

Table 1 lists the header files that are included with Interix in the /usr/include directory and compares them with the header files found in the Linux and Solaris variants.

Table 1. Interix header files in /usr/include, with Linux and Solaris variants

Header	Interix	Linux	Solaris
ar.h	X	X	X
assert.h	X	X	X
cpio.h	X	X	X
ctype.h	X	X	X
curses.h	X	X	X
db.h	X	X	
dirent.h	X	X	X
dlfcn.h	X	X	X
err.h	X	X	**sys/err.h**
errno.h	X	X	X
eti.h	X	X	X
excpt.h	X		
fcntl.h	X	X	X
Features.h	X	X	
float.h	X		X
Fnmatch.h	X	X	X
form.h	X	X	X

fts.h	X	X	
ftw.h	X	X	X
glob.h	X	X	X
grp.h	X	X	X
libgen.h	X	X	X
limits.h	X	X	X
locale.h	X	X	X
malloc.h	X	X	X
math.h	X	X	X
Memory.h	X	X	X
menu.h	X	X	X
mpool.h	X		
ndbm.h	X		X
netdb.h	X	X	X
new.h	X		
nl_types.h	X	X	X
nl_types_private.h	X		
panel.h	X	X	X
paths.h	X	X	
poll.h	X	X	X
pty.h	X	X	
pwcache.h	X		
pwd.h	X	X	X
regex.h	X	X	X
search.h	X	X	X
setjmp.h	X	X	X
signal.h	X	X	X
stdarg.h	X		X
stddef.h	X		X
stdio.h	X	X	X
stdlib.h	X	X	X
string.h	X	X	X
strings.h	X	X	X
stropts.h	X	X	X
syslog.h	X	X	X
tar.h	X	X	X
term.h	X	X	X
Termios.h	X	X	X
time.h	X	X	X
tzfile.h	X		X
ucontext.h	X	X	X
ulimit.h	X	X	X
unctrl.h	X	X	X
unistd.h	X	X	X
utime.h	X	X	X
utmpx.h	X	X	X
va_list.h	X		

	X		X
vis.h	X		
wait.h	X	X	X
wchar.h	X	X	X
xti.h	X		X

Table 2 lists the header files that are included with Interix in the /usr/include/sys directory and compares them with the header files found in the Linux and Solaris variants.

Table 2. Interix header files in /usr/include/sys, with Linux and Solaris variants

Header	Interix	Linux	Solaris
cdefs.h	X	X	
dir.h	X	X	
endian.h	X		
errno.h	X	X	X
fault.h	X		X
fcntl.h	X	X	X
file.h	X	X	X
fsid.h	X		X
ioctl.h	X	X	X
ipc.h	X	X	X
mkdev.h	X		X
mman.h	X	X	X
msg.h	X	X	X
param.h	X	X	X
procfs.h	X	X	X
queue.h	X	X	
reg.h	X	X	X
regset.h	X		X
resource.h	X	X	X
select.h	X	X	X
sem.h	X	X	X
shm.h	X	X	X
siginfo.h	X		X
signal.h	X	X	X
socket.h	X	X	X
stat.h	X	X	X
statvfs.h	X	X	X
stropts.h	X	X	X
syscall.h	X	X	X
syslog.h	X	X	X
Termios.h	X	X	X
time.h	X	X	X
timeb.h	X	X	X
times.h	X	X	X
types.h	X	X	X

ucontext.h	X	X	X
uio.h	X	X	X
un.h	X	X	X
user.h	X	X	X
utsname.h	X	X	X
wait.h	X	X	X

Processes

The UNIX and Windows process models are very different. However, because these differences are hidden by the Interix subsystem, it is possible to migrate UNIX code to Interix with few process code modifications.

The following sections confirm the similarities between UNIX and Interix process functions and highlight the few areas that need to be changed when migrating code.

Creating a New Process

Interix supports the UNIX process creation APIs, **fork()** and **vfork()**. Code that uses these calls does not require any modifications to compile under Interix.

For an example of code that ports to Interix without change, see "Creating a Process in UNIX by Using fork and exec" in Chapter 9, "Win32 Code Conversion."

Replacing a Process Image (exec)

Because Interix supports all six **exec** calls—**exec()**, **execl**, **execle()**, **execlp()**, **execv()**, **execve()** and **execvp()**—code that uses these calls does not need to be modified.

However, if the **exec** call is used with a **setuid()** call, the code must be modified. In this case, replace the **exec()** and **setuid()** combination with an Interix **exec*_asuser()** call.

> **Note** In Interix, **man setuid** produces the **setuid** manual page.

For more detailed information, see the section "The exec*_asuser Functions" under "Users, Groups and Security," later in this chapter.

For an example of code that ports to Interix without change, see "Replacing a Process Image in UNIX Using exec" in Chapter 9, "Win32 Code Conversion."

Process Hierarchy

In UNIX, processes have a parent-child relationship. This hierarchical arrangement is used to manage processes within applications.

The Win32 subsystem does not use a process hierarchy.

However, Interix does maintain a process hierarchy, and even tracks the parent-child relationship of Win32 processes on the same system. The Interix command **ps-efi** displays processes and their relationship to each other. (The **-i** option is specific to Interix and shows the process hierarchy.)

The following shows sample output from **ps-efi**, showing Interix processes followed by Win32 processes:

```
         UID  PID   PPID    STIME   TTY  TIME     CMD
...
    joeuser   2049  1       Jun  4  n00  0:00.59  /bin/csh -l
    joeuser   9545  2049    06:31:58     n00      0:00.01 ps -ef
...
+SYSTEM  8    0     Jun  4  S00     1:14.33  SystemProcess
+SYSTEM  152  8     Jun  4  S00     0:00.86    \SystemRoot\System32\smss.exe
+SYSTEM  176  152   Jun  4  S00     3:53.04      C:\WINNT\system32\csrss.exe C:
+SYSTEM  1040 152   Jun  4  S00     0:23.39      C:\WINNT\system32\psxss.exe C:
+SYSTEM  196  152   Jun  4  S00     0:24.47      C:\WINNT\system32\winlogon.exe
+SYSTEM  236  196   Jun  4  S00     0:48.41        C:\WINNT\system32\lsass.exe
+SYSTEM  224  196   Jun  4  S00     0:47.00        C:\WINNT\system32\services.ex
```

Waiting for a Child Process

Interix supports most of the wait-for-process-termination calls, including **wait()** and **waitpid()**. It does not support calls in the style of Berkley Software Distribution (BSD). When BSD-style wait calls are used, modify code to use the suggested equivalents in Interix, as shown in Table 3.

Table 3. BSD-style calls and Interix equivalents

Function name	Description	Suggested Interix replacements
wait3('status, options, NULL)	Waits for process termination	**waitpid (-1, 'status, options)**
wait3('status, options, 'r_usage)	Waits for process termination	**cpid = waitpid (-1, 'status, options)** **getrusage (cpid, 'r_usage)**
wait4(pid, 'status, options, NULL)	Waits for process termination	**waitpid (pid, 'status, options)**
wait4(pid, 'status, options, 'r_usage)	Waits for process termination	**waitpid (pid, 'status, options)** **getrusage (pid, 'r_usage)**

The functions supported by Interix are defined by the POSIX and UNIX standards and will be more portable than the older forms they replace.

Combining **waitpid()** with **getrusage()** doesn't produce the same results as **wait3()** or **wait4()** without taking some additional steps in the ported application. The idea is to capture **getrusage (RUSAGE_CHILDREN, . . .)** information at some instant before the child process has terminated; once the child terminates and has been waited for, capture a second set of **getrusage (RUSAGE_CHILDREN, . . .)** information and compute the difference between the data contained in the two structures.

Managing Process Resource Limits

As stated in the Chapter 9, Win32 Code Conversion, the UNIX **getrlimit** function returns the process resource limits, **getrusage** returns current usage and **setrlimit** sets new limits. Interix supports these three functions and the common limit names. The common limit names are shown in Table 4.

Table 4. Process resource limit names

Limit	Description

RLIMIT_CORE	Maximum size, in bytes, of a core file created by this process. If the core file will be larger than RLIMIT_CORE, the write is terminated at this value. If the limit is set to 0, then no core files are created.
RLIMIT_CPU	Maximum CPU time, in seconds, that a process can use. If the process exceeds this time, the system generates SIGXCPU for the process.
RLIMIT_DATA	Maximum size, in bytes, of a process data segment. If the data segment grows larger than this value, the functions **brk**, **malloc** and **sbrk** fail.
RLIMIT_FSIZE	Maximum size, in bytes, of a file created by a process. If the limit is 0, the process cannot create a file. If a write or truncate call exceeds the limit, further attempts fail.
RLIMIT_NOFILE	Highest possible value for a file descriptor, plus one. This limits the number of file descriptors a process can allocate. If more than RLIMIT_NOFILE files are allocated, functions allocating new file descriptors can fail and generate the error EMFILE.
RLIMIT_STACK	Maximum size, in bytes, of a process stack. The stack will not automatically grow past this limit. If a process tries to exceed the limit, the system generates the SIGSEGV error for the process.
RLIMIT_AS	Maximum size, in bytes, of total available memory for a process. If this limit is exceeded, the memory functions **brk**, **malloc**, **mmap** and **sbrk** fail with errno set to ENOMEM, and automatic stack growth fails as described for RLIMIT_STACK.

Some other resource limit names that are sometimes used in UNIX code are not available in Interix. These names are shown in Table 5. For these names, modify code to use the suggested replacements shown in the table.

Table 5 Process resource limit names not available in Interix

Limit	Description	Suggested Interix replacement
RLIMIT_MEMLOCK	Maximum locked-in-memory address space, in bytes.	Interix has no mechanism for determining or enforcing limits on this resource.
RLIMIT_NPROC	Maximum number of processes.	The only Interix equivalent that provides programmatic information on process limits is **sysconf**(_SC_ CHILD_MAX) is the only Interix equivalent that provides programmatic information on process limits, but this is not an exact equivalent.
RLIMIT_RSS	Maximum resident set size, in bytes, of address space in a process's address space in bytes.	Interix has no mechanism for determining or enforcing limits on this resource.
RLIMIT_VMEM	Maximum size, in bytes, of mapped address space in a	Interix has no mechanism for determining or enforcing limits on

	process's mapped address space in bytes. If this limit is exceeded, the brk and **mmap** functions fail with **errno** set to ENOMEM. In addition, the automatic stack growth fails as described for RLIMIT_STACK.	this resource.

Process Groups

Functions in this category provide support for the management of processes as a group. Because the functions in this group are not supported by Interix, code must be modified to use the recommended replacement functions shown in Table 6.

Table 6. Process group functions not supported by Interix

Function name	Description	Suggested Interix replacement
Getpgid(0)	Gets process group ID of the calling process.	**getpgrp()**
Getpgid(pid)	Gets process group ID for process PID.	Replace with the **getpgid(pid)** function (see the description paragraph that follows this table).
Setpgrp()	Sets process group ID of the calling process.	**setpgid(0,0)**
Tcgetsid	Gets process group ID for session leader for controlling terminal.	**struct utmpx *getutxid (const struct utmpx *id)**

As noted, Interix does not support the **getpgid(pid)** function, which returns the process group ID for a given process. This information can be obtained, but you would need to use the /proc mechanism, which allows a program to retrieve a variety of information about any running process. An implementation of such a function might look like this:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

extern int errno;

pid_t getpgid(pid_t pid)
{
    char procfile[25];
    char stat_rec[40];
    char inbuf[110];
    char field1[10], field2[100];

    FILE *in;

    sprintf(procfile, "/proc/%d/stat", pid);

    in = fopen(procfile, "r");
    if (in == NULL) {
        errno = ESRCH;       /* No such process */
        return(-1);
    }
```

```
//Scan file for "pgid" entry

    while(fgets(inbuf, sizeof(inbuf), in))
    {
        sscanf(inbuf, "%s\t%s\n", field1, field2);
        if (strcmp(field1,"pgid") == 0)
            return( (pid_t) atoi(field2));
    }

    errno = ENOSYS;     /* Function not implemented */
    return(-1);
}
```

Process Management

Functions in this category provide support for the scheduling and priority management of processes. These functions are **getpriority()**, **setpriority()** and **nice()**. These functions operate on a "nice" value, an integer in the range of -20 to +20, where a nice of -20 means that the process has the highest possible priority. Interix maps nice values to Windows process scheduling priorities according to the following rules:

- A nice value of 0 corresponds to the default Windows scheduling priority 10.
- Positive nice values are applied as a reduction in Windows scheduling priority; for example, assigning a process a nice value of +4 would result in the process being given a Windows scheduling priority of 6.
- Negative nice values are applied as an increase in Windows scheduling priority; a process nice value of -4 would cause the process to have a Windows scheduling priority of 14.

Regardless of nice value, the lowest Windows priority Interix applies to a process is 1; the highest Windows priority assigned by Interix is 30. Microsoft recommends that no process be assigned a priority higher than 15—that is, a nice value of -5. The Interix subsystem itself runs at a Windows priority of 15. Setting a higher priority on any application yields unpredictable results.

Any Interix process can lower the Windows priority of any process owned by the same user (that is, increase its nice value). The effective user of a process must have been granted the SE_INC_BASE_PRIORITY_NAME Windows privilege to increase the Windows scheduling priority of any process owned by the same user (that is, decrease it's nice value). The effective user of a process must have been granted the SE_TCB_NAME Windows privilege to affect any process owned by any other user.

Signals and Signal Handling

There have been four different implementations of signals in the history of UNIX. The Interix Software Development Kit (SDK) supports only the POSIX.1 set of signal semantics. However, the Interix SDK does support several different sets of signal-handling APIs, as follows:

- American National Standards Institute (ANSI) C signals are supported by the function **signal()**. While this function behaves exactly as required by the ANSI C standard, that definition leaves windows of time in which signals may be lost or mishandled. The **sigaction()** API allows applications to close those windows.
- POSIX.1 signals are supported by the functions sigaction(), sigpending(), sigprocmask(), sigsuspend (), sigemptyset(), sigfillset(), sigaddset(), sigdelset() and sigismember().
- BSD 4.3 signals are supported by the functions **killpg()**, **sigsetmask()**, **sigblock()** and **sigvec()**. The signal mask for these functions is **int**, not **sigset_t**.

(Although **sigpause()** is provided by the Interix SDK, it is provided as the System V call, which does not behave in the same way as the BSD call.)

If a future release of the Interix SDK supports more than 32 signals, these functions will become obsolete. Therefore, rather than depend on these functions, it is better to convert code to use the POSIX.1 signal calls.

- System V signals are supported with the functions **sighold()**, **sigignore()**, **sigpause()**, **sigrelse()** and **sigset()**.

Table 7 lists the POSIX-supported signals, all of which are supported by Interix.

Table 7. POSIX-supported signals

Signal name	Description	Default action/effect	Number
SIGABRT	Abnormal termination	Terminate process	6
SIGALRM	Time-out alarm	Terminate process	14
SIGBUS	Bus error	Terminate process	10
SIGCHLD	Change in status of child	Ignore	18
SIGCONT	Continues stopped process	Ignore	25
SIGFPE	Floating-point exception	Terminate process	8
SIGHUP	Hang up	Terminate process	1
SIGILL	Illegal hardware instruction	Terminate process	4
SIGINT	Terminal interrupt character	Terminate process	2
SIGIO	I/O completion outstanding	Ignore	19
SIGKILL	Termination	Terminate process (cannot be caught or ignored)	9
SIGPIPE	Write to pipe with no readers	Terminate process	13
SIGPOLL	Pollable event (Sys V) - synonym of SIGIO	Ignore	22
SIGPROF	Profiling timer alarm	Terminate Process	29
SIGQUIT	Terminal quit character	Terminate Process	3
SIGSEGV	Invalid memory reference	Terminate Process	11
SIGSTOP	Stop process	Stop process (cannot be caught or ignored)	23
SIGSYS	Invalid system call	Terminate process	12
SIGTERM	Software termination	Terminate process	15
SIGTRAP	Trace trap	Terminate process	5
SIGTSTP	Terminal stop character	Stop process	24
SIGTTIN	Background read from control TTY	Stop process	26
SIGTTOU	Background write to control TTY	Stop process	27

SIGURG	Urgent condition on socket	Ignore	21
SIGUSR1	User-defined signal	Terminate process	16
SIGUSR2	User-defined signal	Terminate process	17
SIGVTALRM	Virtual time alarm	Terminate process	28
SIGXCPU	CPU time limit exceeded	Terminate process	30
SIGXFSZ	File size limit exceeded	Terminate process	31

Interix supports most of the signal handling functions. However, Interix does not support some non-standard platform-specific implementations, such as **sigfpe**, signal handling for specific SIGFPE codes. Table 8 shows platform-specific functions not supported by Interix and Interix substitutes, if they exist.

Table 8. Platform-specific signal functions not supported by Interix

Function name	Description	Suggested Interix replacement
bsd_signal	Simplified signal facilities.	See sample code in the section "UNIX bsd_signal Code Replacement," immediately following this table.
getcontext	Gets current user context.	No support or equivalent in Interix.
Gsignal	Software signals.	No support or equivalent in Interix.
makecontext	Manipulates user contexts.	No support or equivalent in Interix.
Psiginfo	Software signals,	No support or equivalent in Interix.
Psignal	System signal messages.	No support or equivalent in Interix.
setcontext	Sets current user context.	No support or equivalent in Interix.
sig2str	Translates the signal number *signum* to the signal name.	Write a simple table lookup routine. (see Table 10.7.)
sigaltstack	Sets or gets signal alternative stack context.	No support or equivalent in Interix.
Sigfpe	Handles signals for specific SIGFPE codes.	**_controlfp**(. . . ,_MCW_EM).
siggetmask	Gets the current set of masked signals.	Use **sigblock(mask)** or **sigsetmask (mask)** with mask set to zero (0).
siginterrupt	Allows signals to interrupt functions.	Controlled by the SA_RESTART flag passed to sigaction().
Sigsend	Sends a signal to a process or a group of processes.	No support or equivalent in Interix.
sigsendset	Sends a signal to a process or a group of processes.	No support or equivalent in Interix.
Sigstack	Sets and/or gets alternative signal stack context.	No support or equivalent in Interix.
Ssignal	Software signals.	No support or equivalent in Interix.
str2sig	Translates the signal name *str* to a signal number.	Write a simple table lookup routine. (See Table 10.7.)
swapcontext	Manipulates user contexts.	No support or equivalent in Interix.
sys_siglist	System signal messages.	Implement the signals shown in Table 10.7 as a vector of signal message strings.

UNIX bsd_signal Code Replacement

Code that uses the **bsd_signal()** function should be implemented by using other signal functions in Interix.

The function call **bsd_signal(sig, func)** can be implemented as follows:

```
#include <signal.h>
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
         return(SIG_ERR);
    return(oact.sa_handler);
}
```

The code above can support calls to **bsd_signal** in a migrated application; it can also be used to replace use of **signal()** in BSD-derived applications as long as the signal handler expected a single parameter of type *int*. If the handler expected any other parameter or parameters, the use of **signal()** must be modified to use **sigaction()**.

Threads

Interix does not support threads. Rewrite code that uses threads to use processes instead of threads or rewrite the application for Win32. Chapter 4, Assessment and Analysis, recommended that threaded code should not be migrated to Interix because it takes significant effort to rewrite the code to use processes. Because ease of migration is what motivates most migrations to Interix, it is better to put that effort into a rewrite to Win32.

Because Interix doesn't support threads, it also doesn't provide the reentrant-safe function variants defined by the UNIX standards. These functions are named for the non-reentrant version of the same function but have **_r** appended to their names—for example, although Interix supports **getpwent()**, it does not support **getpwent_r()**. In the absence of threads, all of the reentrant-safe functions can be replaced by their non-reentrant versions. The conversion is straightforward.

Memory Management

Because Interix supports the majority of UNIX memory management calls, porting code by using memory management is generally straightforward. However, because there are a few specific differences, this section discusses how to address the differences in your code.

Heap

Interix supports most memory functions. The platform-specific memory management functions shown in Table 9 are not supported and code will need to use the alternative functions.

Table 9 Platform-specific memory management functions

Function name	Description	Suggested Interix replacement
Alloca	Memory allocator (from stack frame of caller).	Use **malloc(size_t size)** and **call free()** because space is not automatically freed on return. Potential performance issues are possible because alloca allocated memory on the stack frame of the caller instead of the heap.
Cfree	Debugging memory allocator.	free(void *ptr)
getpagesize	Gets system page size.	Always returns 65536 (64K) regardless of the actual Windows page size. The **getconf (_SC_PAGE_SIZE)** and **sysconf (_SC_PAGE_SIZE)** functions also always return 65536 (64K).
mallocctl	MT hot memory allocator.	Not supported. There may be open source versions of other allocators, which can be used.
Mallopt	Provides for controls over the allocation algorithm.	None; the supported **malloc()** has no controllable options **//mallopt**
memalign	Debugging memory allocator.	No support or equivalent in Interix. No support is given, or equivalent is possible, in Interix.
Valloc	Equivalent to memalign(sysconf (_SC_PAGESIZE),size).	malloc(size_t size)
watchmalloc	Debugging memory allocator.	No support or equivalent in Interix. No support is given, or equivalent is possible, in Interix.

Memory-Mapped Files

Interix supports memory-mapped files by using the **mmap** function. The length of the mapped space, in bytes, is rounded up to the nearest multiple of **sysconf(_SC_PAGE_SIZE)**. This means that the value returned by **sysconf(_SC_PAGE_SIZE)** (or **sysconf(_SC_PAGESIZE)**) is not the virtual-memory page size used by the system, but the value used by the **mmap** call. Code should work without modification unless it assumes page sizes to be smaller than 64 KB; most applications are written without any assumption regarding page size.

For an example of UNIX code that ports to Interix without change, see "Memory Mapped Files" in Chapter 9, Win32 Code Conversion.

Shared Memory

Shared memory permits two or more processes to share a region of memory. Data is not copied as part of the communication process. Instead, the same physical area of memory is accessed by both the client and the server. Because of this, shared memory performance is considered the best of all interprocess communication (IPC) methods.

Interix supports all of the System V IPC mechanisms, including the shared memory routines **shmat**, **shmctl**, **shmdt** and **shmget**.

The command-line interfaces **ipcs** and **ipcm** are also provided for the management of shared memory segments. The **ipcs** interface reports the status of interprocess communication objects. The **ipcm** interface removes an interprocess communication identifier, such as a shared memory segment.

For an example of UNIX code that ports to Interix without change, see "Shared Memory" in Chapter 9, Win32 Code Conversion.

Synchronizing Access to Shared Resources

Code that uses shared memory must ensure that the processes accessing shared memory are not attempting to access the shared memory resource simultaneously. This is particularly troublesome if one or both are writing to the same shared memory area. To address this, UNIX provides the semaphore object.

But there are two sets of functions for semaphores. The POSIX real-time extensions are used for thread synchronization. The System V semaphores are commonly used for process synchronization.

The POSIX real-time extensions and threads are not supported by Interix.

Interix supports all of the System V semaphores, including the shared memory routines **semctl**, **semget** and **semop**.

The command-line interfaces **ipcs** and **ipcm** are also provided for the management of semaphore objects. The **ipcs** interface reports the status of interprocess communication objects. The **ipcm** interface removes an interprocess communication identifier, such as a shared memory segment.

Users, Groups and Security

The UNIX and Windows security models are quite different. Interix, as a subsystem of Windows, uses the underlying Windows security model. At the same time, Interix attempts to present the Windows security model in a way consistent with UNIX security.

This results in some key differences between the way in which Interix security works and the way standard UNIX security works. (For a discussion of some of these differences, see "Comparison of Windows and UNIX Architectures" in Chapter 2, "UNIX and Windows Compared.") This section covers the differences in the security model and how to modify code to operate under Interix.

The key areas that are addressed here are:

- User names in Interix
- UNIX UIDs and GIDs versus Windows SIDs
- The passwd file structure
- User and group operation functions
- Running programs as other users or groups
- User accounting database functions

User Names in Interix

User names are handled differently in UNIX and Interix. In UNIX, they are lowercase text stored in the **passwd** file. In Interix, the user names are taken from the account database and can contain information about the domain the account is in as well as its user name.

It's important to note that Windows places user and group names in the same namespace, while UNIX places them in separate namespaces. What this means is that a UNIX environment might have a user named *tools* and a group named *tools,* while a Windows environment would forbid that and require, for example, that the group be named *tools_group.* Applications should be written in such a way that user and group names aren't hard-coded; obviously, any code that expects identical user and group names will need to be changed.

From a programming standpoint, functions that use the user or group name in UNIX, such as **getpwnam()** and **getgrnam()**, accept a *DomainName*+*UserName* pair in Interix. The *DomainName* part is explained in more detail below. Any application code that assumes that a user or group name cannot contain a plus sign must be changed. Similarly, applications should be prepared to accept user or group names that are considerably longer than the norm on UNIX systems. (Traditional UNIX systems limit user and group names to eight characters each.)

Interix user names and Windows domains

A domain is a collection of networked Windows-based computers that use a common security database. Most Windows-based computers have access to at least two domains: the network domain, which has a centralized security database; and a local domain, which uses the security database on the local computer. On domain controllers, the network and local domains are the same.

Windows user and group names use the format *DomainName*+*UserName*. The **pw_name** and **gr_name** members in the Interix **passwd** and **group** structures use the same format. They can also be passed in the form +*UserName* or *UserName*.

Table 10 describes how domain and user names are interpreted.

Table 10. Domain and user names in Interix

Name	Description
DomainName+*UserName*	The group or user name belonging to the specified domain.
+*UserName*	When passed to an Interix function, finds the closest matching name in the same search order used by Windows. Usually, this is used with well-known names such as +SYSTEM or +Administrator, and the function will find the matching name on the local computer. However, if the name is not found on the local computer, the function will search the primary domain and other trusted domains for that name, returning the first match. When this format is returned from an Interix function, it indicates a name from the local computer domain.
UserName	The group or user name in the same domain as the calling process.

The Interix principal domain is normally the domain to which the system itself belongs. This can be overridden by changing a registry setting (HKEY_LOCAL_MACHINE\Software\Microsoft\Services For Unix\PrincipalDomain). Applications can determine the system's principal domain through the **getpdomain()** function.

UNIX UIDs and GIDs vs. Windows SIDs

UNIX systems translate user and group names to integers, UID and GID, respectively. These values are typically 32-bit numbers and are assigned arbitrarily by an administrator. It is possible to reuse UID and GID values, and it is possible for multiple user or group names to correspond to a particular UID or GID. The UID and GID values are taken from distinct spaces—that is, it is possible for user *joes* to have UID 556 and group *joeteam* to have GID 556. There is no guarantee of uniqueness for UID or GID values between separate systems. Two UNIX systems can have different human users each with the same UID of 556.

Windows systems translate user and group names to SIDs, which are large (typically 112-bit) structured values. SIDs are automatically assigned by Windows when users or groups are created. SIDs are guaranteed to be globally unique and can never be reused. There is a one-to-one relationship between a SID and a user or group. SIDs for users and groups come from the same space.

The UNIX APIs expose user and group identities as integers of type UID_t and GID_t, respectively. Interix conforms to this requirement by mapping Windows SIDs to UID and GID values. All systems belonging to the same domain will perform this mapping in the same way. Under normal circumstances, there will be no collisions amongst mapped UID and GID values—that is, no two Windows SIDs will be converted to the same UID_t or GID_t.

Because mapped UID and GID values may differ between systems joined to different domains, applications that need to preserve identity information should store user or group names rather than UID or GID values.

The passwd File Structure

The files **/etc/passwd** and **/etc/groups** do not exist on an Interix system; therefore, some of the information contained in the structure **struct passwd** is obtained in a nonstandard way, such as the following:

- **pw_gecos**

  The user information member contains the text from the **Description** field in the Windows user account information.

- **pw_shell**

  The user's shell member is always **/bin/sh**.

Because other applications in both the Interix subsystem and the Win32 subsystem can use the information in the **pw_gecos** and **pw_shell** fields, the contents of the **pw_gecos** member will not necessarily continue to be taken from the **Description** field found in a Windows user account.

Interix also returns two additional parameters not available on UNIX systems:

- The **pw_change** parameter returns the time until the password must be changed.
- The **pw_expire** parameter returns the time when the user's account expires.

User and Group Operation Functions

Because the Interix security model is integrated with the Windows security subsystem, Interix does not support all of the user and group functions that manipulate or obtain information from

the **/etc/passwd**, **/etc/group**, **/etc/shadow**, or **/etc/gshadow** files. Table 11 summarizes the calls that Interix does not support and recommended replacements.

Table 11. User and group calls not supported by Interix

Function name	Description	Suggested Interix replacement
Endspent	Indicates that the caller expects to do no further shadow password retrieval operations.	**endspent()**
Fgetgrent, fgetgrent_r	Gets the group file entry.	**struct group *getgrent()**
fgetpwent, fgetpwent_r	Gets the password file entry.	**struct passwd *getpwent()**
Fgetspent, fgetspent_r	Reads from a Shadow file stream.	**struct passwd *getpwent()** Note: Interix does not support a concept of a shadow password file.
getgrent_r, getgrgid_r, getgrnam_r	Gets the group file entry.	**struct group *getgrent (void)** **struct group * getgrgid (GID_t GID)** **struct group * getgrnam (const char *groupname)**
Getpw	Gets the password file entry from UID.	**getpwuid(UID) or getpwnam (*login)** Note: This conversion should be performed before the port to Interix on all supported UNIX platforms.
getpwent_r, getpwnam_r, getpwuid_r	Gets the password file entry.	**getpwent, getpwnam, getpwuid** Note: The re-entrant routines are not required on Interix at this time because it is not multithreaded.
getspent, getspent_r	Gets the shadow password file entry.	**struct passwd *getpwent()** Note: Interix does not support a concept of a shadow password file, and the re-entrant routines are not required on Interix at this time because it is not multithreaded.
Getspnam, getspnam_r	Gets the shadow logon name file entry.	**getpwnam(*login)** Note: Interix does not support a concept of a shadow password file, and the re-entrant routines are not required on Interix at this time because it is not multithreaded.
initgroups(*name, GID_t basegid)	Initializes the supplementary group access list.	**getgroups (int gidsetsize, GID_t grouplist[])** For a usage example, see the sample code in "Interix initgroups Example" below.
lckpwdf, ulckpwdf	Manipulates the shadow password database lock file.	**//lckpwdf** **//ulckpwdf**
Putpwent	Writes a password file entry.	For password change: **chpass(*fq_user, *oldpw, *newpw)**

Putspent	Writes a shadow password file entry.	For password change: **chpass(*fq_user, *oldpw, *newpw)**
setgroups	Sets supplementary group access list Ids.	The list of supplementary groups can be retrieved with **getgroups**, but it is not possible to build and apply an arbitrary list of supplementary groups.
setresgid, where rgid=egid=sgid	Sets real, effective, and saved group ID.	**setgid (GID)** For more information about changing UID and GID in Interix, see the "Changing User ID by Using the Interix setuser() Function" section in this chapter.
setresgid, where rgid!=egid! =sgid	Sets real, effective, and saved group ID.	**setregid (rgid, egid)** For more information about changing UID and GID in Interix, see the "Changing User ID by Using the Interix **setuser()** Function" section in this chapter.
setresuid, where ruid=euid=suid	Sets real, effective, and saved user ID.	**setuid (UID)** For more information about changing UID and GID in Interix, see the "Changing User ID by Using the Interix **setuser()** Function" section in this chapter.
setresuid, where ruid!=euid! =suid	Sets real, effective, and saved user ID.	**setreuid (ruid, euid)** For more information about changing UID and GID in Interix, see the "Changing User ID by Using the Interix **setuser()** Function" section in this chapter.
Setspent	Sets the shadow password entry in the database.	Interix does not support a concept of a shadow password file.

Interix initgroups example

```
<initgroups.c>
```

Interix getgroups conversion example

```
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>

int main()
{
   struct group *ptr_group;
   GID_t grouplist[100];
   int num;

   num = getgroups (99, grouplist);
```

```
   printf("group\tID:\n---------------\n");
   if (num > 0)
      while(num--) {
         ptr_group = getgrgid(grouplist[num]);
         printf("%s\t%d\n", ptr_group->gr_name, grouplist[num]);
      }
   else
      printf("num=%d\n", num);

   exit(0);
}
```

Running Programs as Other Users or Groups

Some programs require access to resources to which the current user has no access. This is solved on UNIX systems by running the program as a specific user or group rather than as the current user or group. UNIX programs can use special permissions on the program's executable file called the **setuid** and **setgid** bits.

According to the POSIX standard, a file's permissions include bits to set a UID (**setuid**) and to set a GID (**setgid**). If either or both bits are set on a file and a process executes that file, the process runs with an identity based on the UID or GID of the file respectively.

When you run a program that has the **setuid** bit set in the file permissions, it runs as if the file's owner has executed it no matter who actually started it. These user and group ids are called *effective user ID* and *effective group ID*.

When an Interix process executes a file that has the **setuid** or **setgid** bit set, Interix constructs local security tokens for the process with the privileges assigned to the owner (if **setuid** is set) and/or group (if **setgid** is set) of the file. Because these tokens are local, they are not recognized by other computers on the network. Even if the file is owned by a member of the Domain Admins group, the process still does not have trusted access to other computers in the domain by using Windows networking.

For example, a process executes a program file that has its **setuid** bit set and that is owned by a member of the Domain Admins group. If that program attempts to change a domain user's password, that attempt fails because the security tokens of the process are local and not recognized by other systems in the domain. If, on the other hand, the program attempts to change a local user's password, the attempt succeeds because the file's owner is a member of the Domain Admins group, which typically belongs to the local computer's Administrators group.

Because of this, a program that has the **setuid** bit set in the file permissions runs as if the file's owner had executed it, no matter who actually started it. These user and group IDs are called *effective user ID* and *effective group ID*.

On UNIX systems, the user and/or group owner is often set to root, thereby allowing a non-root program to run with root privileges. On Interix, the owner should be set to the Local Administrator or to a member of the Domain Admins group.

In summary, UNIX and Interix systems maintain at least two user and group Ids: the effective user ID and effective group ID, and the real user ID and real group ID. Most UNIX systems and Interix also support a saved set user ID and saved set group ID.

Changing between real and effective user ID in setUID programs

Often in programs with the **setuid** bit set, it is desirable to be able to change the real and effective IDs during execution. For example, programmers might want to do one of the following in **setuid** programs:

- Change from the initial effective user/group ID to the real user/group ID
- Change from the real user/group ID back to the initial effective user/group ID

These two actions can be performed by using the UNIX **setuid()** function. In addition to this, Interix provides another function called **setuser()**, which simplifies changing the User ID.

Changing user ID by using the Interix setuser() function

The proprietary Interix **setuser()** function changes the effective and real UID and GID of the current process to that of the specified user name. All of the security attributes and permissions become those of the specified user name. The current working directory of the process does not change.

To use setuser() to replace the standard UNIX calls

1. Add the following include statement to the code:

   ```
   #include <interix/security.h>
   ```

2. Rewrite the code to use the **setuser()** function as defined below:

   ```
   int setuser(char *username, char *password, int flags)
   ```

   Set the arguments as follows:
   - The *username* argument is the name of the user. If the user name is not fully qualified with a domain name (that is, *domain*+*name*), it is not changed. On a system configured for workgroups, specify the domain as NULL: that is, use the string **"+name"** for *username*.
   - The *password* argument is the plaintext password for the specified user name.
   - The *flags* argument comprises control flags. Possible values for *flags* are defined in interix/security.h as follows:

     SU_COMPLETE changes the real and effective user and group IDs and all security attributes to the default for the specified user.

     SU_CHECK verifies that the process can perform a **setuser()** action by using **SU_COMPLETE** for the specified user name and password. This is a quick way to verify a password for a user. This action is equivalent to the older Interix call, authenticateuser().

The **setuser()** function has the advantage of changing both the UID and the GID with only one function call. However, it has the disadvantage that it is only supported on Interix.

There can be performance degradation if a process changes identity to a user who does not have permission to be located in the current working directory. The best solution to this is to call **chdir()** to a directory known to be permitted for the new identity after the call to **setuser()**.

The Interix exec*_asuser() functions

Interix provides a set of interfaces used to execute a process as another user. These functions and structures are defined in the header file security.h. They include **execl_asuser()**, **execle_asuser()**,

execlp_asuser(), **execv_asuser()**, **execve_asuser()** and **execvp_asuser()**.

All of these functions use a user security structure, **struct usersec**, to identify another user. The **usersec** structure is illustrated below:

```
struct usersec {
   char *   user;
   char *   domain;
   char *   password;
   int      logontype;
   int      logonprovider;
};
```

The **domain**, **user**, and **password** members of the structure have their normal meanings. The **logontype** and **logonprovider** members are provided for future development; leave them at the default values of 0.

The six **exec*_asuser()** functions correspond to the six **exec** functions: **execl()**, **execle()**, **execlp()**, **execv()**, **execve()** and **execvp()**. The arguments after the **struct usersec** member are identical to their **exec** counterparts.

User Accounting Database Functions

Interix supports a subset of the routines used to track and manage a user accounting database. Interix does support the following calls: **endutxent**, **getutxent**, **getutxid**, **getutxline**, **pututxline** and **setutxent**.

Table 12 shows user account database functions that are not supported in Interix, and recommended alternatives to them:

Table 12. User account database functions not supported by Interix

Function name	Description	Suggested Interix replacement
Acct	Enables or disables process accounting.	No support or equivalent in Interix.
Endutent	Closes the currently open database.	**endutxent**
Getutent	Extracts the next entry from a UTMP database.	**getutxent**
Getutid	Searches forward from the current point in the UTMP database.	**getutxid**
Getutline	Searches forward from the current point in the UTMP database.	**getutxline**
Getutmp	Copies the information stored in the members of the UTMPX structure to the corresponding members of the UTMP structure.	No support or equivalent for the UTMP structure in Interix.
Getutmpx	Copies the information stored in the members of the UTMP structure to the corresponding members of the UTMPX structure.	No support or equivalent for the UTMP structure in Interix.
Logwtmp	Appends an entry to the WTMP file.	**pututxline**

Pututline	Writes the supplied UTMP structure into the UTMP database.	**pututxline**
Setutent	Resets the input stream to the beginning.	**setutxent**
Updwtmp	Appends an entry to the WTMP file.	**pututxline**
updwtmpx	Writes the contents of the UTMPX structure pointed to by UTMPX to the database.	**pututxline**
utmpname	Changes the name of the database file examined to another file.	
utmpxname	Changes the name of the database file examined from **/var/adm/utmpx** to any other file, typically **/var/adm/wtmpx**.	

File and Data Access

Interix has some differences with UNIX file and data access because of the underlying Windows input/output system. Consequently, certain UNIX features are different or do not work in Interix.

This section discusses:

- Differences in Interix and UNIX file I/O calls
- The Interix **ioctl()** function implementation
- Directory operations
- File system operations in Interix

Differences in Interix and UNIX File Input/Output

Interix does not support file I/O with memory caching turned off (O_DIRECT), and it also does not support the file I/O APIs listed in Table 13. Table 13 also provides code to implement the missing functions, if they are needed.

Table 13 File I/O APIs not supported by Interix

Function	Description	Suggested Interix replacement

pread(fd, *buf, nbytes, offset)	Reads from a file descriptor at a given offset.	`ssize_t pread(int fd, void *buf, size_t count, off_t offset)`
		`{`
		`  ssize_t size;`
		`  off_t rtn`
		`  if (lrtn=seek(fd, offset, SEEK_SET) < 0)`
		`  return( (ssize_t) rtn );`
		`  size = read(fd, *buf, nbytes)`
		`  return(size);`
		`}`

pwrite (fd, *buf, nbytes, offset)	Writes to a file descriptor at a given	`ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)`
		`{`
		`  ssize_t size;`
		`  off_t rtn`
		`  if (lrtn=seek(fd, offset, SEEK_SET) < 0)`

offset.

```
        return( (ssize_t) rtn );
    size = write(fd, *buf, nbytes)
    return(size);
}
```

The Interix ioctl() Function Implementation

The **ioctl()** interface has many uses. The POSIX.1 committee did not standardize the **ioctl()** interface because the last argument cannot be type-checked: its type depends upon the request. Instead, the committee assigned pieces of functionality to other interfaces.

In other words, **ioctl()** does not have a single standard. Its arguments, returns, and semantics vary according to the device driver. The call is used for operations that do not cleanly fit the UNIX stream I/O model.

The **ioctl()** interface has historically been used to handle the following:

- File control (see "File Control and ioctl()" in this section)
- Socket control (see "Socket Control and ioctl()" in this section)
- Disk labels
- Magnetic tape control
- Terminal control

The disk label and magnetic tape I/O requests are not supported in the Interix environment.

The SFU 3.0 API set contains only a few **ioctl()** operations, including window re-sizing. The following two sections explain some additional operations.

File control and ioctl()

The only **ioctl()** requests defined for file control are FIONREAD, to get the number of bytes available to read, and FIONBIO, to set and unset non-blocking I/O.

The FIOCLEX and FIONCLEX requests (usually found in Filio.h) are not provided. They can be replaced with the **fcntl()** FD_CLOEXEC request, as shown in the following example:

```
#ifndef __INTERIX
(void) ioctl(fd, FIOCLEX, NULL)
#else
(void) fcntl(fd, F_SETFL, fcntl(fd, F_GETFD) | FD_CLOEXEC));
#endif
```

Socket control and ioctl()

The only **ioctl()** request defined for sockets in Interix is SIOCATMARK. Other socket control requests are handled by using **fcntl()** or by using functions such as **setsockopt()**.

Directory Operations

Interix supports a subset of the routines used to access directory entries. Interix does not support the calls shown in Table 14.

Table 14. Directory operations routines not supported by Interix

Function name	Description	Suggested Interix replacement
Alphasort	Can be used as the comparison function for the **scandir()** function to sort the directory entries into alphabetical order.	Use **readdir()** to retrieve all entries, then sort as desired using the libc sort() or any other sorting function
Getdents(fd, struct dirent *dirp, count)	Gets directory entries.	**struct dirent * readdir (DIR *dirp)**
getdirentries	Getd directory entries in a file system system-independent format.	**struct dirent * readdir (DIR *dirp)**
Scandir	Scans a directory for matching entries.	**readdir** For more information, see the code example in the "Interix readdir Conversion Example" section below

Replacing scandir in Interix

Because Interix does not support **scandir**, convert to using **readdir**. The example "UNIX scandir Example" below shows UNIX code using the **scandir** function and the following "Interix readdir Conversion Example" example shows how this code is converted to use **readdir** under Interix.

UNIX scandir example

This example lists the contents of the current directory, demonstrating the use of the **scandir** function in UNIX.

```
#include <dirent.h>
main()
{
    struct dirent **namelist;
    int i, n;

    n = scandir(".", &namelist, 0, 0);
    if (n < 0)
        perror("scandir");
    else {
        for (i=0; i<n; i++) {
            printf("%s\n", namelist[i]->d_name);
            free(namelist[n]);
        }
        free(namelist);
    }
}
```

Interix readdir conversion example

In Interix, you must change from using the **scandir** function to using the **readdir** function. The following code shows how to do this. In the **scandir** example, **scandir** is called once and returns the number of directory entries and a structure containing those entries. In contrast, the **readdir** function is called in a loop. Each time **readdir** is called, it returns a new directory entry until there are none left, and then it returns NULL.

```
#include <dirent.h>
#include <stdio.h>
main()
{
    DIR *dp;
    struct dirent *entry;

    if((dp = opendir(".")) == NULL)
        perror("opendir");
    else {
        while((entry = readdir(dp)) != NULL)
            printf("%s\n",entry->d_name);

        closedir(dp);
    }
}
```

For other examples of UNIX code that ports to Interix without change, see "Directory Scanning" in Chapter 9, Win32 Code Conversion.

Working directory

Interix does not support the routines to get the current working directory shown in Table 15.

Table 15. Working directory routines not supported by Interix

Function name	Description	Suggested Interix replacement
get_current_dir_name	Gets current working directory **(define:__USE_GNU)**.	**getcwd(buf, max)**
Getwd	Gets current working directory **(define:__USE_BSD)**.	**getcwd(buf, max)**

The **getwd** API as defined on BSD systems is particularly dangerous, as it is vulnerable to buffer overrun attacks. Replace it with **getcwd** and a buffer of known size.

For an example of UNIX code that ports to Interix without change, see "Working Directory" in Chapter 9, Win32 Code Conversion.

File System Operations in Interix

Operations on file systems in Interix differ in a number of ways from file system operations under UNIX. Some functions are not supported, such as **sync**, **sysfs** and **ustat**. Others have different parameters or use a different set of options for UNIX.

Table 16 lists the file system information functions that need to be replaced.

Table 16. File system information functions not supported by Interix

Function	Description	Suggested Interix replacement
statfs, fstatfs	Get file system statistics.	**statvfs** For more information on statvfs, see Table 17.
Sync	Writes all information in memory	No support or equivalent in Interix.

	that should be on disk, including modified super blocks, modified inodes, and delayed block I/O.	
Sysfs	Gets file system type information.	**statvfs** For more information on statvfs, see Table 17.
Ustat	Gets file system statistics.	Port to **statfs** first.

When using the **statvfs** function in Interix, be aware that the **statfs** structure has some members different from those usually found in UNIX. Table 17 summarizes these differences. In general, references to the **statfs** structure can by replaced with references to the **statvfs** structure.

Table 17. Differences between UNIX statfs and Interix statvfs

UNIX statfs structure	Interix statvfs structure	Description
long f_type	**unsigned long f_type;**	Type of file system. For more information, see "Commonly Supported File Systems in UNIX" and "Supported File System Types in Interix" below.
long f_bsize	**unsigned long f_bsize;**	Transfer block size.
long f_blocks	**unsigned long f_blocks;**	Total data blocks in file system.
long f_bfree	**unsigned long f_bfree;**	Free blocks in file system.
long f_bavail	**unsigned long f_bavail;**	Free blocks available to non-superuser.
long f_files	**lunsigned ong f_files;**	Total file nodes in file system. (Currently returns 0.)
long f_ffree	**unsigned long f_ffree;**	Free file nodes in file system. (Currently returns 0.)
fsid_t f_fsid	**unsigned long f_fsid;**	File system ID.
long f_namelen	**unsigned long f_namemax;**	Maximum length of file names.
long f_spare[6]	**unsigned long f_flag;**	Bit mask of values describing the file system.
	unsigned long f_frsize;	Fundamental block size.
	unsigned long f_favail;	Total number of file serial numbers available to a non-privileged process. (Currently returns 0.)
	unsigned long f_iosize;	Optimal transfer block size.
	char f_mntonname [MNAMELEN+1];	Mountpoint for the file system.
	char f_mntfromname [MNAMELEN+1];	Mounted file system.

When using **statvfs**, the file system types supported by Interix differ from those supported by most implementations of UNIX. The two lists below illustrate this by comparing a common subset of UNIX-supported file systems with those supported in Interix. It is rare for these differences to have any impact on migrating an application.

Commonly supported file systems in UNIX

- AFFS_SUPER_MAGIC 0xADFF
- EXT_SUPER_MAGIC 0x137D
- EXT2_OLD_SUPER_MAGIC 0xEF51
- EXT2_SUPER_MAGIC 0xEF53
- HPFS_SUPER_MAGIC 0xF995E849
- ISOFS_SUPER_MAGIC 0x9660
- MINIX_SUPER_MAGIC 0x137F /* orig. minix */
- MINIX_SUPER_MAGIC2 0x138F /* 30 char minix */
- MINIX2_SUPER_MAGIC 0x2468 /* minix V2 */
- MINIX2_SUPER_MAGIC2 0x2478 /* minix V2, 30 char names */
- MSDOS_SUPER_MAGIC 0x4d44
- NCP_SUPER_MAGIC 0x564c
- NFS_SUPER_MAGIC 0x6969
- PROC_SUPER_MAGIC 0x9fa0
- SMB_SUPER_MAGIC 0x517B
- XENIX_SUPER_MAGIC 0x012FF7B4
- SYSV4_SUPER_MAGIC 0x012FF7B5
- SYSV2_SUPER_MAGIC 0x012FF7B6
- COH_SUPER_MAGIC 0x012FF7B7
- UFS_MAGIC 0x00011954
- XFS_SUPER_MAGIC 0x58465342
- _XIAFS_SUPER_MAGIC 0x012FD16D

Supported file system types in Interix

- ST_FSTYPE_UNKNOWN 0 /* unknown */
- ST_FSTYPE_NTFS 1 /* NTFS */
- ST_FSTYPE_OFS 2 /* OFS-NT object FS */
- ST_FSTYPE_CDFS 3
- ST_FSTYPE_CDROM ST_FSTYPE_CDFS
- ST_FSTYPE_ISO9660 ST_FSTYPE_CDFS
- ST_FSTYPE_FATFS 4 /* MS-DOS FAT FS */
- ST_FSTYPE_MSDOS ST_FSTYPE_FATFS
- ST_FSTYPE_HPFS 5 /* OS2 HPFS */
- ST_FSTYPE_SAMBA 6 /* Samba FS */
- ST_FSTYPE_NFS 8 /* NFS */
- ST_FSTYPE_MAX 8 /* for now */

File system mount entry management

Interix does not support dynamically mounted file systems, as UNIX does. Therefore, Interix does not include a file to define the mount table.

Mount tables are stored in different files on different implementations of UNIX. They are usually stored under the /etc directory and have names such as mtab and fstab or mnttab and vfstab.

This is unlikely to impact the migration of an application. However, if the application does use dynamically mounted file systems, remove the functionality and ensure that the file system is permanently mounted.

Library porting

Applications still use the gdbm database (that is, an indexed file storage system), which is good at storing relatively static indexed data. The following APIs are used by an application that uses the gdbm database:

- **gdbm_close()**
- **gdbm_delete()**
- **gdbm_exists()**
- **gdbm_fdesc()**
- **gdbm_fetch()**
- **gdbm_firstkey()**
- **gdbm_nextkey()**
- **gdbm_open()**
- **gdbm_reorganize()**
- **gdbm_setopt()**
- **gdbm_store()**
- **gdbm_strerror()**
- **gdbm_sync()**

An example of application code using these functions follows:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <gdbm.h>
#include <string.h>

#define TEST_DB_FILE "/tmp/gdbmtest1"
#define NUM_ITEMS 3

struct test_data {
    char some_chars[10];
    int  an_integer;
    char more_chars[21];
};

int main() {
    struct test_data data_to_store[NUM_ITEMS];
    struct test_data item_retrieved;

    char key_to_use[20];
    int i, result;

    datum key_datum;
    datum data_datum;

    GDBM_FILE gdbm_ptr;

    gdbm_ptr = gdbm_open(TEST_DB_FILE, 512, O_RDWR | O_CREAT, 0666,
0);
    if (!gdbm_ptr) {
        fprintf(stderr, "Database Open Failed\n");
        exit(EXIT_FAILURE);
    }

/* put some data in the structures */
    memset(data_to_store, '\0', sizeof(data_to_store));
    strcpy(data_to_store[0].some_chars, "First!");
    data_to_store[0].an_integer = 47;
```

```
    strcpy(data_to_store[0].more_chars, "Who?");
    strcpy(data_to_store[1].some_chars, "Second");
    data_to_store[1].an_integer = 13;
    strcpy(data_to_store[1].more_chars, "What?");
    strcpy(data_to_store[2].some_chars, "Third");
    data_to_store[2].an_integer = 3;
    strcpy(data_to_store[2].more_chars, "Where?");

    for (i = 0; i < NUM_ITEMS; i++) {
/* build a key to use */
        sprintf(key_to_use, "%c%c%d",
            data_to_store[i].some_chars[0],
            data_to_store[i].more_chars[0],
            data_to_store[i].an_integer);

/* build the key datum structure */
        key_datum.dptr = (void *)key_to_use;
        key_datum.dsize = strlen(key_to_use);
        data_datum.dptr = (void *)&data_to_store[i];
        data_datum.dsize = sizeof(struct test_data);

        result = gdbm_store(gdbm_ptr, key_datum, data_datum,
GDBM_REPLACE);
        if (result != 0) {
            fprintf(stderr, "gdbm_store failed on key %s\n",
key_to_use);
            exit(2);
        }
    } /* for */

/* now try to retrieve some data */
    sprintf(key_to_use, "SW%d", 13); /* this is the key for the
second item */
    key_datum.dptr = key_to_use;
    key_datum.dsize = strlen(key_to_use);

    data_datum = gdbm_fetch(gdbm_ptr, key_datum);
    if (data_datum.dptr) {
        printf("Data retrieved\n");
        memcpy(&item_retrieved, data_datum.dptr, data_datum.dsize);
        printf("Retrieved item - %s %d %s\n",
               item_retrieved.some_chars,
               item_retrieved.an_integer,
               item_retrieved.more_chars);
    }
    else {
        printf("No data found for key %s\n", key_to_use);
    }

    gdbm_close(gdbm_ptr);

    exit(EXIT_SUCCESS);
}
```

An attempt to compile this application with the Interix **gcc** compiler yields output similar to the following:

```
% gcc -o dbmtest1 dbmtest1.c -ldb
dbmtest1.c:5: gdbm.h: No such file or directory
```

Porting the GNU gdbm database to Interix

It is possible to modify the program listed in "Library Porting" to use the older dbm database, which is supported by Interix. But a better approach that is relatively easy to implement is to port the GNU gdbm database library to Interix.

To port the gdbm database to Interix

1. Obtain the gdbm compressed tar archive from the GNU Web site.
2. Unzip and extract the archive files into the Interix source tree, for example, at /usr/examples:

```
$ gunzip < gdbm-1.8.0.tar.gz | tar xf-
$ cd gdbm-1.8.0
$ ls configure
configure
```

3. Run the resulting configure file.

   The script created during the building of the Interix development environment to run configuration scripts will be used.

   The following is an example of the listing that results. The configuration looks good except that it did not detect Interix's ability to create shared (dynamic) libraries. For further information about creating shared libraries in Interix, see "Building Applications with Interix," in Chapter 7, Creating the Development Environment.

```
$ cat /usr/local/bin/runconfig
set -x
CPPFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
CXXFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
CFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
LDFLAGS="-L/usr/local/lib" \
./configure --prefix=/usr/local \
--host=intel-pclocal-interix $*
$ runconfig
+ ./configure --prefix=/usr/local --host=intel-pclocal-interix
+ CPPFLAGS=-D_ALL_SOURCE -I/usr/local/include CXXFLAGS=-
D_ALL_SOURCE -I/usr/local/include CFLAGS=-D_ALL_SOURCE \
-I/usr/local/include LDFLAGS=-L/usr/local/lib
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc -D_ALL_SOURCE -
I/usr/local/include -L/usr/local/lib) works... yes
checking whether the C compiler (gcc -D_ALL_SOURCE -
I/usr/local/include -L/usr/local/lib) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking how to run the C preprocessor... gcc -E
checking for a BSD compatible install... /bin/install -c
checking host system type... intel-pclocal-interix
checking for ranlib... :
checking for ld used by GCC... /usr/contrib/bin/ld
checking if the linker (/usr/contrib/bin/ld) is GNU ld... yes
checking for BSD-compatible nm... /bin/nm -B
checking whether ln -s works... yes
checking for gcc option to produce PIC... -fPIC
checking if gcc PIC flag -fPIC works... yes
checking if gcc static flag -static works... -static
checking if the linker (/usr/contrib/bin/ld) is GNU ld... yes
```

```
checking whether the linker (/usr/contrib/bin/ld) supports shared
libraries... no
checking command to parse /bin/nm -B output... no
checking how to hardcode library paths into programs...
unsupported
checking for /usr/contrib/bin/ld option to reload object files...
-r
checking dynamic linker characteristics... no
checking if libtool supports shared libraries... no
checking whether to build shared libraries... no
checking whether to build static libraries... yes
checking for objdir... .libs
creating libtool
checking for working const... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for sys/file.h... yes
checking for unistd.h... yes
checking for fcntl.h... yes
checking for sys/types.h... yes
checking for memory.h... yes
checking for main in -ldbm... no
checking for main in -lndbm... no
checking for rename... yes
checking for ftruncate... yes
checking for flock... yes
checking for bcopy... yes
checking for fsync... yes
checking for getopt... yes
checking for ANSI C header files... yes
checking for off_t... yes
checking for st_blksize in struct stat... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
creating autoconf.h
```

4. Run the **make** command.

```
$ make
```

You should see output similar to the following results:

```
/bin/sh ./libtool --mode=compile gcc -c   -I. -I. -O dbminit.c
gcc -c -I. -I. -O dbminit.c
  .
  .
: .libs/libgdbm.a
creating libgdbm.la
(cd .libs && ln -s ../libgdbm.la libgdbm.la)
```

5. Now, install the gdbm libraries, man pages, info pages and commands.

```
$ make install
```

You should see output similar to the following results:

```
/mkinstalldirs /usr/local/lib /usr/local/include
/usr/local/man/man3 /usr/local/info
```

```
mkdir /usr/local/man/man3
mkdir /usr/local/info
/bin/sh ./libtool install -c libgdbm.la /usr/local/lib/libgdbm.la
install -c libgdbm.la /usr/local/lib/libgdbm.la
install -c .libs/libgdbm.a /usr/local/lib/libgdbm.a
: /usr/local/lib/libgdbm.a
chmod 644 /usr/local/lib/libgdbm.a
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/lib
```

To link against installed libraries in a given directory, LIBDIR, you must use the -LLIBDIR flag during linking.

6. You will also need to do one of the following:

```
See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
/bin/install -c -m 644 -o bin -g bin gdbm.h
/usr/local/include/gdbm.h
install: unknown group bin
*** Error code 1

Stop.
```

In this setup the installation of the header file gdbm.h failed because Interix does not have the standard UNIX bin user or group. The following steps resolve this.

7. To ensure that the header file gdbm.h is installed you can either:
   - Run the following:

     ```
     $ /Table 10/install -c -m 644 -o COMPUTERNAME+Administrator -g
     +Administrators gdbm.h /usr/local/include/gdbm.h
     ```

   - Or change the Makefile, before running: **make install**

     Modify the Makefile to the following:

     ```
     #TABLE 10 = bin
     TABLE 10 = COMPUTERNAME+Administrator
     #TABLE 10 = bin
     TABLE 10 = +Administrators
     ```

     where COMPUTERNAME is the name of the computer that the **make install** command is being executed on.

     You can determine COMPUTERNAME by entering "hostname" (without the quotes) at a command prompt.

     Now try to build the application again:

     ```
     $ Table 10 -o dbmtest1 dbmtest1.c -
     I/usr/local/include-lgdbm
     $ ./Table 10
     ```

You should see output similar to the following results:

```
Table 10 retrieved
Table 10 item - Second 13 What?
Table 10!
```

Interprocess Communication

Interix supports all the various forms of Interprocess Communication (IPC). The forms most familiar to UNIX developers are discussed in the following sections:

- Ordinary or Anonymous pipes
- Named pipes
- Message queues
- System V IPC mechanisms

Ordinary (Anonymous) Pipes

Process pipes are supported under Interix by using the standard C runtime library. Interix supports all the pipe function calls, including **popen**, **pclose** and **pipe**. There is no need to change any references to these calls in your code.

Pipes are frequently used between UNIX processes to connect the standard output file descriptor of one process to the standard input file descriptor of a second process, causing the results of the first program to be treated as the input data of the second. This sequence of commands is called a "pipeline".

This mechanism works unchanged under Interix when connecting Interix processes. It is possible to use this same mechanism to connect an Interix process to a Win32 process that it creates. In nearly all cases, everything works as is without any change. Problems with using pipes to communicate between Interix and Win32 processes generally fall into two categories:

- Line termination character. Interix defines a line as ending with the \n character; Win32 defines lines as ending with the \r\n sequence. Some applications are sensitive to the precise line termination sequence. Use the **flip** command in the pipe to change line termination as necessary.
- End Of File (EOF) handling when attempting serial use of a pipe. Once a Win32 process has closed a pipe it is writing to, it is not possible for an Interix application to serially use that pipe. For example, this command will display only the contents of *file1*:

  ```
  (cmd.exe /c type file1; cat file2) | cat
  ```

  By introducing a second pipeline using the **cat32** utility, this problem can be solved using this command:

  ```
  (cmd.exe /c type file1; cat file2 | cat32) | cat
  ```

Named Pipes (FIFOs)

Interix also supports named pipes. These are also referred to as First in First Out (FIFO). A named pipe is a special type of that is created in the file system (with the **mknod** or **mkfifo** function calls or command line programs), but behaves like process pipes.

Interix supports the two function calls for creating a named pipe, **mknod** and **mkfifo**. If possible, use **mkfifo** for making FIFO special files because it is more portable.

Again, it is not necessary to modify code that uses these functions for it to compile under Interix.

These named pipes are distinct from, and not interoperable with, the identically named Win32 interprocess communication mechanism. The only way to use Win32 named pipes to communicate between Interix and Win32 processes is through ordinary pipes, as described above.

For examples of UNIX code that ports to Interix without modification, see the following sections in Chapter 9 , Win32 Code Conversion:

- "Creating a Named Pipe"
- "Opening a FIFO"
- "Interprocess Communication with FIFOs"

Message Queues

Message queues are very similar to named pipes, but there is no need to open and close pipes. Interix supports all the message queue routines, **msgctl**, **msgget**, **msgrcv**, and **msgsnd**. Code that uses these functions does not need to be modified.

System V IPC mechanisms

Interix includes support for the System V IPC shared memory and semaphore mechanisms. (For more information, see the discussion in "Memory Management" in this chapter.) Code that uses these mechanisms should not need changes to compile under Interix.

For an example of UNIX code that ports to Interix without modification, see "Memory Mapped Files" in Chapter 9, Win32 Code Conversion.

For an example of UNIX shared memory code that ports to Interix without modification, see "Shared Memory" in Chapter 9, Win32 Code Conversion.

Sockets and Networking

The Interix Software Development Kit (SDK) implements BSD-style socket interfaces, including **bind()**, **accept()**, and **connect()**. The Interix SDK implementation uses the Windows Winsock library to access the network. This means that TCP/IP sockets and all of the installed Winsock protocols are supported.

For more information about Winsock, see the Winsock documentation.

Exceptions to socket support in Interix are discussed in the following sections:

- Host name to address translation
- Network groups
- Network socket calls
- Transport level interface calls

Host Name to Address Translation

Interix does not support the generic transport name-to-address translation routines. Interix does support the **gethostby** routines, except for the re-entrant versions (that is, routines with the **_r** suffix). Table 18 shows other host or domain name **get** or **set** routines not supported by Interix.

Table 18. Host name translation routines not supported by Interix

Function name	Description	Suggested Interix replacement
getdomainname	Gets the NIS domain name.	No equivalent in Interix. The principal Windows domain for the system can be obtained through **getpdomain()**.
Gethostid	Gets the unique identifier of the current host.	No equivalent in Interix, but should be a very rare occurrence in any application except a network administration application.
gethostbyaddr_r, gethostbyname_r, gethostent_r	Re-entrant "safe" versions.	Replace with equivalents lacking the _r suffix. Interix is not multithreaded, and therefore does not need to support the re-entrant versions.
setdomainname	Sets the NIS domain name.	No equivalent in Interix.
Sethostid	Sets the unique identifier of the current host.	No equivalent in Interix, but should never need to be set. It is restricted to the root user account.
sethostname	Sets the name of the host machine. (This call is restricted to the superuser and is normally used only when the system is booted.)	No equivalent in Interix.

Network Groups

Interix does not currently support the network group APIs. However, it does support network group designations in hosts.equiv and .rhosts files.

Network Socket Calls

These functions constitute the BSD sockets library, libsocket.a. This library is not automatically linked by the C compilation system, so the **-lsocket** option must be included on the **gcc** or **cc** command line to link with this library.

Interix supports all the socket calls except those summarized in Table 19 and Table 20.

Table 19. Socket calls not supported by Interix

Function name	Description	Suggested Interix replacement
cmsg macros	Access ancillary data	No equivalent in Interix.
freeaddrinfo	Removes address entry from linked list.	Currently no API support for IPv6
freehostent	Removes IP node entry from linked list.	Currently no API support for IPv6

gai_strerror	Helps applications print error messages based on the EAI_* codes returned by **getaddrinfo**.	Currently no API support for IPv6
getaddrinfo	Translates between node name and address.	Currently no API support for IPv6
getipnodebyaddr	Gets IP node entry.	Currently no API support for IPv6
getipnodebyname	Gets IP node entry.	Currently no API support for IPv6
getnameinfo	Translates between node name and address.	Currently no API support for IPv6
inet_ntop	Process network address structures	Currently no API support for IPv6.
inet_pton	Create a network address structure	Currently no API support for IPv6.
rcmd_af	Returns a stream to a remote command and includes support for Ipv6.	Currently no API support for IPv6.
Recvmsg	Receives a message from a socket.	No equivalent in Interix.
rexec_af	Returns a stream to a remote command and includes support for Ipv6.	Currently no API support for IPv6.
Rresvport_af		Currently no API support for IPv6.
Sendmsg	Sends a message to a socket.	No equivalent in Interix.

The two sockets functions **recvmsg** and **sendmsg** appear in many network applications but are not supported by Interix. These functions are the only way to pass an open file descriptor from one running process to another running process.

For examples of UNIX code using sockets that port directly to Interix, see "Sockets and Networking" and "Sockets Example" in Chapter 9, Win32 Code Conversion.

Each re-entrant interface performs the same operation as its non-re-entrant counterpart. The only difference is the **_r** suffix. The re-entrant interfaces, however, use buffers supplied by the caller to store returned results, and they are safe for use in both single-threaded and multithreaded applications. If the application is not multithreaded, then the **_r** routines can be safely replaced by removing the **_r** suffix and the additional parameters.

Table 20. Interix replacements for re-entrant routines

Function name	Description	Suggested Interix replacement
getnetbyaddr_r	Searches for a network entry with the network address.	**getnetbyaddr**
getnetbyname_r	Searches for a network entry with specified name.	**getnetbyname**
getnetent_r	Enumerates network entries from the database.	**getnetent**
getprotobyname_r	Sequentially searches from the beginning of the file until a matching protocol name is found, or EOF is encountered.	**getprotobyname**
getprotobynumber_r	Sequentially searches from the beginning of the file until a matching protocol number is	**getprotobynumber**

	found, or EOF is encountered.	
getprotoent_r	Gets a matching protocol name.	**getprotoent**
getservbyname_r	Returns a pointer to an object containing the information from a network services database.	**getservbyname**
getservbyport_r	Returns a pointer to an object containing the information from a network services database.	**getservbyport**
getservent_r	Returns a pointer to an object containing the information from a network services database.	**getservent**

Transport Level Interface (XTI) Calls

The XTI APIs, defined by The Open Group's **X/Open Transport Interface** specification, define protocol-independent networking functions similar to those provided by the old SVR4 TLI (Transport Level Interface) APIs. XTI, and TLI before it, were primarily used as interfaces to the ISO OSI protocol family or to the STREAMS networking stack. In general, use of XTI should be replaced by use of the more standard BSD Sockets interface.

Interix support for XTI is limited solely to the functions and features required to access the UDP Internet protocol. Interix does not support some extended calls (see Table 21), which are mainly used with "expedited data" and the management or configuration of variables and parameters.

Table 21. Transport level interface calls not supported by Interix

Function name	Description	Suggested Interix replacement
nlsgetcall	Gets client data passed via the listener.	No equivalent in Interix.
nlsprovider	Gets the name of the transport provider.	No equivalent in Interix.
nlsrequest	Formats and sends a listener service request message.	No equivalent in Interix.
t_rcvv	Receives data or expedited data sent over a connection and puts data into one or more noncontiguous buffers greater than or equal to.	No equivalent in Interix.
t_rcvvudata	Receives a data unit in one or more noncontiguous buffers.	No equivalent in Interix.
t_sndv	Sends data or expedited data from one or more noncontiguous buffers on a connection.	No equivalent in Interix.
t_sndvudata	Sends a data unit from one or more noncontiguous buffers.	No equivalent in Interix.
t_sysconf	Gets configurable XTI variables.	No equivalent in Interix.

The Process Environment

Some of the key elements in the process environment differ between UNIX and Interix. This section discusses these key elements and how you implement them in Interix:

- Environment variables
- Using stdarg and varargs
- Temporary files
- Computer information
- Logging system messages

Environment Variables

An environment block is a block of memory allocated within the process address space. Each block contains a set of name value pairs. All UNIX variants support process environment blocks. The particular differences between Interix and other UNIX variants depend on which UNIX variant is being ported to Interix. For example, some UNIX variants do not support either the **setenv** or **unsetenv** function calls, whereas Interix does.

There are usually no issues in porting calls to environment variable functions to Interix. However, when porting System V Interface Definition (SVID) code, instead of the process environment being defined as a third argument to **main()**, it is defined as **extern char **environ**. To modify the environment for the current process, use **getenv()** and **putenv()**. To modify the environment to be passed to a child process, use **getenv()**, **setenv()**, and **putenv()**, or build a new environment and pass it to the child by using the **envp** argument of **exec()**.

For an example of UNIX code using these functions that ports to Interix without modification, see "Environment Variables" in Chapter 9, Win32 Code Conversion.

Using stdarg and varargs

Rather than conflict with the historical routines in varargs.h, the International Standards Organization/American National Standards Institute (ISO/ANSI) C standard defines stdarg.h, a new mechanism for dealing with variable argument lists. The **varargs** mechanism uses a magic name, **va_alist**, for the first argument in a list; **stdarg** uses the last required argument. This means that **stdarg** must have at least one named parameter. The Interix SDK ships both headers, so there is no need to convert from one to the other. For ANSI standard code, convert from **varargs** to **stdarg**.

Usually, it is possible to translate easily from **varargs** to **stdarg** because most functions with variable argument lists have a known first-argument type.

The following examples show how code using **varargs** is rewritten to use **stdarg**. The first example is a trivial, error-printing function that uses the **varargs** mechanism:

```
#include <varargs.h>
printerror( va_alist );
void printerror (va_alist )
va_dcl
{
   va_list ap;
   char *fmt;
   va_start(ap);
   fmt = va_arg(ap, char *);
   vprintf(stderr, fmt, ap);
   va_end(ap);
```

```
}
```

The next example shows how the code is changed when **stdarg** is used to replace **varargs**. Because the function in the previous example uses a format string as its first argument, it can easily be used as the known argument in the function in the following example:

```
#include <stdarg.h>
void printerror (char *fmt, ...)
{
   va_start(ap, fmt);
   vfprintf(stderr, fmt, ap);
   va_end(ap);
}
```

The first argument must be given a name, even when the routine takes a terminated list and no fixed arguments. For example, the following function prints a set of strings, but the first argument is entirely artificial, created to meet the needs of the **stdarg** package:

```
#include <stdarg.h>
pr_str(char *first, ...)
{
   char * current;
   va_list argp;
   va_start(argp,first);
   current = first;
   while (current != NULL){
   fputs(current,stdout);
   current = va_arg(argp, char *);
   }
   va_end(argp);
}
```

The following examples show the use of conditional compilation that uses the ANSI flag to provide backward compatibility.

The original application, **var.c**, passes a variable number of arguments by using **va_start**, **va_arg**, **va_end**, **va_list** and **va_dcl** (UNIX only):

```
/* var.c: The program below illustrates passing a variable
 * number of arguments using the following macros:
 *      va_start            va_arg              va_end
 *      va_list             va_dcl (UNIX only) */
#include <stdio.h>
#include <varargs.h>
int average( va_list );
int main( void )
{
   /* Call with 3 integers (-1 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );

   /* Call with 4 integers. */
   printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );
   /* Call with just -1 terminator. */
   printf( "Average is: %d\n", average( -1 ) );
}
/* Returns the average of a variable list of integers. */
```

```
int average( va_alist )
va_dcl
{
   int count = 0, sum = 0, i ;
   va_list marker;
   i = va_arg(marker, int);
   va_start( marker );
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);
   }
   va_end( marker );              /* Reset variable arguments.
*/
   return( sum ? (sum / count) : 0 );
}
```

When compiled, the following errors display. The errors occur because Interix does not support the pre-ANSI version of **args** available in varargs.h.

```
var.c: In function `main':
var.c:11: warning: passing arg 1 of `average' makes pointer from
integer without a cast
var.c:11: too many arguments to function `average'
var.c:14: warning: passing arg 1 of `average' makes pointer from
integer without a cast
var.c:14: too many arguments to function `average'
var.c:16: warning: passing arg 1 of `average' makes pointer from
integer without a cast
var.c: In function `average':
var.c:20: argument `__builtin_va_alist' doesn't match prototype
var.c:7: prototype declaration
```

To modify the code to work on Interix but still retain backward compatibility with pre-ANSI version, introduce the flag **#ifdef ANSI** and write the ANSI version for **varargs** as listed below:

```
/* var2.c: The program below illustrates passing a variable
 * number of arguments using the following macros:
 *      va_start            va_arg              va_end
 *      va_list             va_dcl (UNIX only) */
#include <stdio.h>
#define ANSI            /* Comment out for Pre-ANSI version     */
#ifdef ANSI             /* ANSI compatible version          */
#include <stdarg.h>
int average( int first, ... );
#else                   /* Pre-ANSI version          */
#include <varargs.h>
int average( va_list );
#endif
int main( void )
{
   /* Call with 3 integers (-1 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );
   /* Call with 4 integers. */
   printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );
   /* Call with just -1 terminator. */
   printf( "Average is: %d\n", average( -1 ) );
}
```

```
/* Returns the average of a variable list of integers. */
#ifdef ANSI             /* ANSI compatible version    */
int average( int first, ... )
#else
int average( va_alist )
va_dcl
#endif
{
   int count = 0, sum = 0, i ;
   va_list marker;
#ifdef ANSI
   i = first;
   va_start( marker, first );     /* Initialize variable arguments.
*/
#else
   i = va_arg(marker, int);
   va_start( marker );
#endif
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);
   }
   va_end( marker );              /* Reset variable arguments.
*/
   return( sum ? (sum / count) : 0 );
}
```

Temporary Files

Interix supports functions that create temporary files. It is not necessary to modify code to migrate these functions to Interix.

For an example of UNIX code using these functions that ports to Interix without modification, see "Temporary Files" in Chapter 9, Win32 Code Conversion.

Computer Information

Interix supports functions that obtain information about the machine the application is executing on. No modifications to ported code are typically required.

This information includes the following:

- Host name
- Operating system name
- Network name of the machine
- Release level of the operating system
- Version number of the operating system
- Hardware platform name

The same information can be obtained by using the **uname -a** Interix shell command.

For an example of UNIX code using these functions that ports to Interix without modification, see "Computer Information" in Chapter 9, Win32 Code Conversion.

Logging System Messages

Interix provides the standard UNIX **syslogd** daemon for storing and rerouting log messages from applications and system services. The Interix **syslogd** daemon handles only Interix processes that were designed to use the **syslog** API. It does not handle log messages from the Win32 subsystem. If **syslogd** is not running, all messages intended for **syslogd** are appended to the file **/var/adm/log/logger**.

The **syslog**, **vsyslog**, **openlog**, **closelog** and **setlogmask** function calls are all supported by Interix with the same set of severity levels, including:

- LOG_ALERT
- LOG_CRIT
- LOG_DEBUG
- LOG_EMERG
- LOG_ERR
- LOG_INFO
- LOG_NOTICE
- LOG_WARNING

There is also a superset of facility indicators, including:

- LOG_AUTH
- LOG_CRON
- LOG_DAEMON
- LOG_KERN
- LOG_LOCAL(0-7)
- LOG_LPR
- LOG_MAIL
- LOG_NEWS
- LOG_USER
- LOG_UUCP

It is not necessary to modify code that uses **syslog** calls on Interix.

For an example of UNIX code using these functions that ports to Interix without modification, see "Logging System Messages" in Chapter 9, Win32 Code Conversion.

Daemons and Services

A UNIX daemon is a process that provides a specific service or services, such as:

- The **inetd** daemon listens for connections on certain Internet sockets.
- The **nfsd** daemon implements the user-level part of the NFS (directory/file sharing) services.
- The **syslogd** daemon provides system utilities that provide support for system logging and kernel message trapping.

On traditional UNIX systems, a daemon is a process that runs for an extended period of time, but does not have a controlling terminal. A Windows service is a background process that is similar to a daemon process. Daemons can be ported to the Interix subsystem by using the service interface.

Many daemons use **setuid()** or **seteuid()** to run as a particular user. This does not work on Windows,

however, because of the way in which Windows security is structured. A daemon invoked by **inetd** does not have these restrictions. It inherits the environment of the **inetd** process. In this context, the term daemon refers to a daemon process invoked by an Interix process that runs in the context of the Interix subsystem. An Interix service is an Interix process that is tied to the Win32 execution environment by the **psxrun.exe** program.

Code for traditional daemons is written differently from code for Windows services. Some of the rules for coding daemons can cause problems if they are applied to writing code for Windows services.

The Interix SDK contains the same **daemon()** interface that is found on BSD systems. Because the **daemon()** API uses **fork()** and **setsid()**, it should be used only in daemons that will be invoked by **inetd** or other master daemons.

> **Note** Interix includes ports of both the **inetd** and **syslogd** daemons and makes them available as Windows services.

Porting a UNIX Daemon to Interix

This section discusses porting a UNIX daemon to Interix and calling that daemon from either **inetd** or another master daemon. Daemons ported in this way cannot be run as a Windows service. If a daemon needs to run as a Windows service, use the instructions in the section "Porting a UNIX Daemon to an Interix Service."

When porting a daemon for UNIX to Interix, the daemon should already have the features described in the list below. For an example, see the code listing for a UNIX (and Interix) daemon in "Example–Interix Daemon Code" later in this chapter.

- A daemon must first use fork to fork and then use exit. The child process created by the fork must then create a session and set the process group ID by calling setsid.

  The **fork()** function is required because the current process is already a process group leader, which would otherwise cause the call to **setsid()** to fail. By using the call to **setsid()** in the child process, it becomes the leader of the new session and the process group leader of the new process group, and it has no controlling terminal. This is necessary for a daemon process. The parent process exits: It is no longer needed because the daemon code runs as the child.

- The signal handling routine (terminate, in this case) is a common characteristic of a daemon process. By using it, the daemon can perform cleanup operations when it receives the SIGTERM signal (for example, when the system shuts down). To use the terminate handling routine, the daemon must call the **signal()** system call, which installs its signal handler for the SIGTERM signal.

Porting this daemon is a simple process. All that is needed is to recompile, relink, and execute the daemon from the command line.

To port the daemon

1. Recompile and relink the UNIX daemon by using **gcc**:

```
$ gcc -o msgd msgd.c
$ ./msgd &
```

2. Compile the test program found in "Test Client for Interix Daemon" by using the following command:

   ```
   $ gcc -o msg msg.c
   ```

3. Launch the test application (**msg**) that will interact with the Interix daemon by entering the following command:

   ```
   $ ./msg
   ```

   Output similar to the following appears. (Input is shown in bold.)

   ```
   Enter some text: what
   msgd received: WHAT
   Enter some text: Where are we?!
   msgd received: WHERE ARE WE!
   Enter some text: end
   msgd received: END
   $
   ```

4. To obtain information about the Interix processes executing on the Windows-based system, enter the following command:

   ```
   $ ps-ef
   ```

   Notice that **./msgd** is listed with a parent process of 1. This is the UNIX **init** process, the parent of all UNIX and Interix processes. Its primary role is to create processes from a script stored in the file **/etc/inittab**. It also controls autonomous processes required by any particular system.

   > **Note** Interix does not use **/etc/inittab** because it only runs at runlevel 2, multi-user, without NFS.

5. Terminate the Interix daemon process by entering the following command:

   ```
   $ kill <pid of ./msgd>
   ```

   (The PID of ./msgd is found under the PID heading.)

The **msgd** daemon is successfully ported to Interix.

The majority of daemons port directly to Interix, other coding issues apart. However, porting a daemon to an Interix service requires some changes, as discussed in "Porting a UNIX Daemon to an Interix Service," below.

Porting a UNIX Daemon to an Interix Service

An Interix daemon has a number of limitations: It can be called only from a master daemon, and it is not integrated into the Windows service mechanisms. When a daemon is converted into a service, it can be managed from the Windows Control Panel **Services** item, as well as from the Interix command line.

Before going into more details on how to convert a daemon into a service, it is important to mention two commands that can help tie Interix processes to the Win32 execution environment:

- The **posix.exe** program starts an Interix process with a controlling terminal.
- The **psxrun.exe** program starts an Interix process without a controlling terminal.

In the case of a Windows service, the process must run without a controlling terminal, so **psxrun** is required.

Interix services can be administered by using the Windows Control Panel **Services** item or by using the **service** utility provided with Interix.

The **service** utility is used to install the service as a particular user and to stop the service. The user name and password must be provided when the service is installed. If no user name is provided, the user defaults to LocalSystem, which provides an administrative login without network access. When a request to kill a particular service comes from either the Windows Control Panel **Services** item or from the **service** utility, **psxrun** sends the signal SIGTERM to the service.

Converting daemon code into Interix service code

To convert daemon code into Interix service code, the daemon code needs the following modifications:

- Ensure that the service exits when it receives the SIGTERM signal.

  It should catch the SIGTERM signal, clean up, and shut down. Ideally, the service should not spend more than a few seconds cleaning up. Otherwise, there can be detrimental interactions with the Service Control Manager, such as lost communications.

- Use the **daemon()** interface to create the service.

  Using the daemon interface makes it easier to create the service because it causes the calling program to fork. Then, the parent exits and the child performs a **setsid()**. This disassociates the process from its current process group, session and controlling terminal. On successful completion of this call, the process is the session leader of a group in which it is the only member, and the session has no controlling terminal.

- Change the code so that it does not fork and then the parent exits.

  If the parent process exits, **psxrun** treats the program as having exited, and the Windows Service Control Manager reports that the service was never successfully started.

- Do not call **setsid( )** to create a new session.

  This does not work because of Windows Security. Use **#ifdef** to skip that code, and then replace it with a simple **exec_asuser()** call, or install the service as a particular user. The **psxrun.exe** program already ensures that the service process is a session leader.

- Do not access network drives from the daemon.

  Network drives are typically mounted on drive letters when a user logs in, then unmounted when that user logs out. A service program cannot depend on a given network drive being mounted on a given drive letter. If a service uses **net.exe** to mount a network drive, the drive letter it uses will become unavailable to interactive users, which may cause **winlogon.exe** to display error messages. If a service must access a network drive, then reserve specific drive letters for exclusive use by the

system.

For an example of these code and compilation steps, see the listing in "Example–Interix Service," where the code in "Example–Interix Daemon Code" has been converted into an Interix service.

Installing a daemon as a service

The **service** utility is required to install a daemon as a service. This utility can also be used to start and stop the service. The user name and password must be provided when the service is installed. If no user name is provided, the user defaults to LocalSystem, which provides an administrative login without network access. When a request comes from either Windows Control Panel **Services** or from the **service** utility to kill a particular service, **psxrun** sends the signal SIGTERM to the service. This is why the proper response to the SIGTERM signal is important.

To install and start the Windows (daemon) service, enter the following commands:

```
$ service install ./wmsgd -s auto
$ service start wmsgd
```

Functions to Change for Interix

This section describes functions that need to be changed or removed before code will compile under Interix. These functions include:

- Math routines. (See "Math Routines" below.)
- Regular expressions. (See "Regular Expressions" below.)
- System and C library and other platform-specific APIs. (See "System/C Library and Miscellaneous APIs" below.)
- Extended UNIX Code (EUC) characters are not supported by Interix and should not be used.
- Wide character-type APIs are not supported by Interix and should not be used.
- Multibyte character-type APIs (ISO/ANSI C and UNIX98) are not supported by Interix and should not be used.
- Long-long character type (64-bit integer) APIs are not supported by Interix and should not be used.
- Message handling APIs are not supported by Interix and should not be used.

Math Routines

There are two sets of mathematical routines not supported by Interix: IEEE floating-point environment control routines and conversion routines.

Interix does not support IEEE floating-point environment control routines, as shown in Table 22.

Table 22. IEEE floating-point environment control routines not supported by Interix

Function	Suggested Interix replacement
Fpclass	No equivalent in Interix.
fpgetmask	No equivalent in Interix.
fpgetround	No equivalent in Interix.
fpgetsticky	No equivalent in Interix.
fpsetmask	No equivalent in Interix.

fpsetround	No equivalent in Interix.
fpsetsticky	No equivalent in Interix.

Interix does not support conversion routines (such as converting decimal record to double), as shown in Table 23.

Table 23. Conversion routines not supported by Interix

Function	Suggested Interix replacement
decimal_to_double	No equivalent in Interix.
decimal_to_extended	No equivalent in Interix.
decimal_to_floating	No equivalent in Interix.
decimal_to_quadruple	No equivalent in Interix.
decimal_to_single	No equivalent in Interix.
double_to_decimal	No equivalent in Interix.
Econvert	No equivalent in Interix.
Extended_to_decimal	No equivalent in Interix.
Fconvert	No equivalent in Interix.
File_to_decimal	No equivalent in Interix.
floating_to_decimal	No equivalent in Interix.
func_to_decimal	No equivalent in Interix.
Gconvert	No equivalent in Interix.
qeconvert	No equivalent in Interix.
Qfconvert	No equivalent in Interix.
qgconvert	No equivalent in Interix.
quadruple_to_decimal	No equivalent in Interix.
seconvert	No equivalent in Interix.
Sfconvert	No equivalent in Interix.
sgconvert	No equivalent in Interix.
single_to_decimal	No equivalent in Interix.
string_to_decimal	No equivalent in Interix.

Regular Expressions

Interix does not support some of the regular expression function calls. However, Table 24 lists functions that can be used to replace them.

Table 24. Regular expression function calls not supported by Interix

Function	Description	Suggested Interix replacement
re_comp	Compiles and executes a regular expression and returns character pointer to NULL on success.	**regcomp(), regexec()**
re_exec	Compiles and executes a regular expression and return integer of 0 or 1 on success.	**regcomp(), regexec()**
Regcmp	Compiles a regular expression and returns pointer to compiled form.	**regcomp()**

Regex	Executes a regular expression.	**regexec()**

System/C Library and Miscellaneous API

Under this category are many specialized, platform-specific APIs, including the following:

- Command-line and shell APIs. (See the section below.)
- String manipulation functions. (See the section below.)
- BSD string and bit functions. (See the section below.)
- Time handling APIs. (See the section below.)
- Other system/C library functions. (See the section below.)
- Kernel calls are not supported by Interix and should not be used.

Command-line and shell APIs

Interix does not support calls to obtain information about legal user shells from the **/etc/shells** file, nor does it support the implementation of the **popt** command-line parser.

Make these changes in Interix for code that uses this feature:

1. Create an **/etc/shells** file containing the legal Interix shells /bin/csh, /bin/ksh, /bin/sh, /bin/tcsh.
2. Add to this file any additional legal shells that have been ported, such as /bin/bash.
3. Write functions called **endusershell**, **getusershell**, and **setusershell**, as described in Table 25.

Table 25. Functions to implement the command-line and shell APIs in Interix

Function name	Description	Suggested Interix replacement
endusershell	Closes the file of legal user shells (/etc/shells).	Create **endusershell** routine to wrap the standard **close()** file routine.
getusershell	Gets legal user shells from /etc/shells.	Create **getusershell** routine to wrap the standard **read()** file routine.
Popt	Parses command-line options	**getopt()**
setusershell	Rewinds the file of legal user shells (/etc/shells).	Create **setusershell** routine to wrap the standard **lseek()** file routine to set /etc/shells file back to beginning.

String manipulation functions

Interix supports most of the standard string handling functions. Interix does not support **atoq**, **memmem**, **stpcpy**, **stpncpy**, **strfmon**, **strfry**, **strnlen** and **strtows**, which need to be replaced as shown in Table 10.26.

Table 10.26 String handling functions not supported by Interix

Function name	Description	Suggested Interix replacement
Memmem	Finds the start of the first occurrence of a substring in the memory area.	**strstr()**

Stpcpy	Copies source string, including the terminating '\0' character, to destination.	**strcpy()**
Stpncpy	Copies at most num characters from source string, including the terminating '**\0**' character, to destination.	**strncpy()**
Strfmon	Formats specified amounts according to the format specification and places the result in a character array of size max.	**sprintf()**
Strfry	Randomizes the contents of string by using **rand()** to randomly swap characters in the string.	Customize **strfry** by using **rand()**.
Strnlen	Returns the number of characters in the string, not including the terminating '**\0**' character, but at most maxlen	**strlen()**

BSD string and bit functions

String interfaces specified by ANSI/ISO C are in String.h. String interfaces found only in the Single UNIX Specification are in Strings.h. The Strings.h file contents are described in Table 27.

Table 27. String interfaces found in the Strings.h file

Function name	Description	Suggested Interix replacement
Bcmp	Compares two strings.	**strcmp()**
Bcopy	Copies at most n characters from **src** string to **dest**.	**strncpy()**
Bzero	Places **len** value of 0 bytes in the string.	**memset (string, 0, len)**
ffs	Finds the first bit set (beginning with the least significant bit) and returns the index of that bit.	Interix supports **ffs**.

The Interix SDK also supports the BSD 4.4 **strsep()** and **strcasestri()** routines.

Time handling APIs

The time functions that are not supported by Interix and that cannot be implemented by some other Interix API or set of API calls are **adjtime**, **adjtimex**, **ntp_adjtime** and **tzsetwall**.

Other system/C library functions

Interix does not support some of the system/C library functions. Table 28 summarizes these functions.

Table 28. System/C library functions not supported by Interix

Function name	Description	Suggested Interix replacement

on_exit	Registers a function to be called at normal program termination.	**atexit(void (*function)(void))** Note: This function does not provide for argument passing as **on_exit** does.
Quotactl	Manipulates disk quotas.	No support or equivalent in Interix.
Stime	Sets time.	No support or equivalent in Interix.

Code Examples

These sections contain code examples to illustrate the migration techniques:

- Test Client for Interix Daemon
- Example–Interix Daemon Code
- Example–Interix Service

Test Client for Interix Daemon

The program used to test the example daemon (**msg**) is listed in this section.

```
$ cat msg.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
    long int my_msg_type;
    char some_text[MAX_TEXT];
};

int main()
{
    int running = 1;
    struct my_msg_st some_data;
    int out_msgid, in_msgid;
    long int msg_to_receive = 0;
    char buffer[BUFSIZ];

/* First, connect to the input message queue. */

    in_msgid = msgget((key_t)1235, 0666);
    if (in_msgid == -1) {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

/* Next, connect to the output message queue. */
    out_msgid = msgget((key_t)1234, 0666);
```

```
    if (out_msgid == -1) {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

    while(running) {
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        some_data.my_msg_type = 1;
        strcpy(some_data.some_text, buffer);

        if (msgsnd(out_msgid, (void *)&some_data, MAX_TEXT, 0) == -1)
{
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }

        if (msgrcv(in_msgid, (void *)&some_data, BUFSIZ,
                msg_to_receive, 0) == -1) {
            fprintf(stderr, "msgrcv failed with error: %d\n", errno);
            exit(EXIT_FAILURE);
        }

        printf("msgd received: %s", some_data.some_text);

        if (strncmp(buffer, "end", 3) == 0) {
            running = 0;
        }
    }

    exit(EXIT_SUCCESS);
}
```

Example–Interix Daemon Code

This example daemon provides an ASCII character conversion service by using one System V (IPC) message queue for input and one for output. It waits for input on the input message queue (1234), converts all lowercase ASCII characters to uppercase, and returns the result to the output message queue (1235).

```
$ cat msgd.c

#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

    int in_msgid, out_msgid;

/* This function reacts to the signal which is passed in the
parameter sig.
    This function is called when a signal occurs.
    This signal is "normally" received during a system shutdown,
```

```
    and this is where cleanup is performed.
    In this case the Input and Output message queues are deleted.
*/
void terminate(int sig)
{
/*  Close the input message queue */
    if (msgctl(in_msgid, IPC_RMID, 0) == -1)
  fprintf(stderr, "msgctl failed (input queue) error: %s\n",
strerror(errno));

/*  Close the output message queue */
    if (msgctl(out_msgid, IPC_RMID, 0) == -1)
  fprintf(stderr, "msgctl failed (output queue) error: %s\n",
strerror(errno));

    exit(0);
}

struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};

/*  The main daemon function must intercept the SIGTERM signal
generated
    for example when the system shuts down.
    Otherwise, it sits in an infinite loop, waiting for messages to
process
    on its input queue
*/
int main()
{
    pid_t pid, sessionID;
    int i;
    int running = 1;
    struct my_msg_st some_data;
    long int msg_to_receive = 0;
    struct msqid_ds buf;

    (void) signal(SIGTERM, terminate);

pid = fork();
    switch(pid)
    {
    case -1:
        fprintf(stderr, "fork failed");
        exit(EXIT_FAILURE);
    case 0:
/* This is the child...so it continues on */

/* Now the child becomes the process group leader
   In general, setsid only fails if the child (i.e. the calling
process)
   is already a process group leader. */

sessionID = setsid();
        if (sessionID == -1) {
            fprintf(stderr, "setsid failed");
            exit(EXIT_FAILURE);
        }
```

```
/* First, set up the input message queue. */

        in_msgid = msgget((key_t)1234, 0666 | IPC_CREAT | IPC_EXCL);

        if (in_msgid == -1) {
            fprintf(stderr, "msgget failed err: %s for input
queue\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
  if (msgctl(in_msgid, IPC_STAT, &buf) != -1)
      printf("Input Queue Permissions are: %o\n", buf.msg_perm.mode);

/* Second, set up the output message queue. */

        out_msgid = msgget((key_t)1235, 0666 | IPC_CREAT | IPC_EXCL);

        if (out_msgid == -1) {
            fprintf(stderr, "msgget failed err: %s for output
queue\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
        if (msgctl(out_msgid, IPC_STAT, &buf) != -1 )
      printf("Output Queue Permissions are: %o\n",
buf.msg_perm.mode);

/* Then the messages are retrieved from the input queue, converted,
and sent
   to the output queue, until an end message is encountered.  */
        while(running) {
            if (msgrcv(in_msgid, (void *)&some_data, BUFSIZ,
                    msg_to_receive, 0) == -1) {
    if (errno != EINTR) {
        fprintf(stderr, "msgrcv failed with error: %s\n",
strerror(errno));
                    exit(EXIT_FAILURE);
    }
    else
        continue;
            }
/*            printf("You wrote: %s", some_data.some_text); */

        i=0;
            while(some_data.some_text[i])
                some_data.some_text[i++] =
toupper(some_data.some_text[i]);

            if (msgsnd(out_msgid, (void *)&some_data, MAX_TEXT, 0) ==
-1) {
                fprintf(stderr, "msgsnd failed with error: %s\n",
strerror(errno));
                exit(EXIT_FAILURE);
            }
        }
        exit(EXIT_SUCCESS);

    default:
/* We're the parent...so just exit*/
exit(0);
    }
 }
```

Example–Interix Service

In the previous code listing, "Example–Interix Daemon Code," the **msgd** daemon was ported to Interix. For this daemon to start at system startup, the daemon must be converted into a service. The changes described and listed in this section are required to convert and install the daemon as a service.

Some of the normal UNIX rules for coding daemons cause problems with the service interface, especially forking, having the parent exit, and then calling **setsid()** to create a new session. Recall that if the parent process has called **exit()**, then **psxrun.exe** treats the program as having exited and the Windows Service Control Manager reports that the service never started successfully. This is what the ported **msgd** code presently does.

To fix this problem, modify the code as follows. Notice that calls to **fork()**, **setsid()**, and the associated logic have been removed from the **wmsgd.c** code.

```
$ cat wmsgd.c

#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

    int in_msgid, out_msgid;

/*  This function reacts to the signal which is passed in the
parameter sig.
    This function is called when a signal occurs.
    This signal is "normally" received during a system shutdown,
    and this is where cleanup is performed.
    In this case the Input and Output message queues are deleted.
*/
void terminate(int sig)
{
/*  Close the input message queue */
    if (msgctl(in_msgid, IPC_RMID, 0) == -1)
  fprintf(stderr, "msgctl failed (input queue) error: %s\n",
strerror(errno));

/*  Close the output message queue */
    if (msgctl(out_msgid, IPC_RMID, 0) == -1)
  fprintf(stderr, "msgctl failed (output queue) error: %s\n",
strerror(errno));

    exit(0);
}

struct my_msg_st {
    long int my_msg_type;
    char some_text[BUFSIZ];
};
```

```
/*  The main daemon function must intercept the SIGTERM signal
generated
    for example when the system shuts down.
    Otherwise, it sits in an infinite loop, waiting for messages to
process
    on its input queue
*/
int main()
{
    int i;
    int running = 1;

    struct my_msg_st some_data;
    long int msg_to_receive = 0;
    struct msqid_ds buf;

    (void) signal(SIGTERM, terminate);

/* First, set up the input message queue. */

    in_msgid = msgget((key_t)1234, 0666 | IPC_CREAT | IPC_EXCL);

    if (in_msgid == -1) {
  fprintf(stderr, "msgget failed err: %s for input queue\n",
strerror(errno));
  exit(EXIT_FAILURE);
    }
    if (msgctl(in_msgid, IPC_STAT, &buf) != -1)
  printf("Input Queue Permissions are: %o\n", buf.msg_perm.mode);

/* Second, set up the output message queue. */

  out_msgid = msgget((key_t)1235, 0666 | IPC_CREAT | IPC_EXCL);

    if (out_msgid == -1) {
  fprintf(stderr, "msgget failed err: %s for output queue\n",
strerror(errno));
  exit(EXIT_FAILURE);
    }
    if (msgctl(out_msgid, IPC_STAT, &buf) != -1 )
  printf("Output Queue Permissions are: %o\n", buf.msg_perm.mode);

/* Then the messages are retrieved from the input queue, converted,
and sent
   to the output queue, until an end message is encountered.  */
    while(running) {

  if (msgrcv(in_msgid, (void *)&some_data, BUFSIZ, msg_to_receive, 0)
== -1) {
      if (errno != EINTR) {
    fprintf(stderr, "msgrcv failed with error: %s\n",
strerror(errno));
    exit(EXIT_FAILURE);
      }
      else
    continue;
  }

  i=0;
  while(some_data.some_text[i]) some_data.some_text[i++] =
```

```
toupper(some_data.some_text[i]);

  if (msgsnd(out_msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
      fprintf(stderr, "msgsnd failed with error: %s\n",
strerror(errno));
      exit(EXIT_FAILURE);
  }

    }
    exit(EXIT_SUCCESS);

}
```

Notice that calls to **fork()**, **setsid()**, and the associated logic have been removed from the **wmsgd.c** code.

To compile and execute the above code

1. To re-compile and re-link the Windows (daemon) service, use the **gcc** command:

   ```
   $ gcc -o wmsgd wmsgd.c
   $ ./wmsgd &
   ```

2. To launch the test application **msg2** to interact with the Windows service, enter the following command:

   ```
   $ ./msg2
   ```

   Note that the output displayed is the same as in "Example–Interix Daemon Code."

3. To terminate the Windows (daemon) service process, use the following command:

   kill PIDofWmsgd

   (The *PID of Wmsgd* is found under the PID heading in a **ps** listing.)

Interix daemons can be installed and executed in two different ways. They must be installed and executed as Windows services if they have to run logged on with a Windows account, such as to access network resources. Otherwise, they can be run as conventional daemons by using the same mechanism as conventional UNIX systems, that is, by creating shell scripts in the /etc/rc2.d/ directory that directly start and stop the daemons. Then the **init** utility can start the daemon when the Interix subsystem initializes.

Because the **wmsgd** daemon does not need to run logged on to a Windows account, it can be installed by creating a shell script in the /etc/rc2.d directory. The following script can be used for the **wmsgd** daemon:

```
#!/bin/sh
#
# /etc/init.d/wmsgd
#
WMSGD=/usr/sbin/wmsgd
PATH=/bin:/usr/contrib/bin
. /etc/init.d/funcs
case $1 in
        start)
                ${WMSGD}
                [ $? = 0 ] && echo "wmsgd started"
```

```
                ;;
        stop)
                killall ${WMSGD}
                [ $? = 0 ] && echo "wmsgd stopped"
                ;;
        *)
                echo "usage: $0 start|stop"
                ;;
        esac
exit 0
```

This script should be created and stored in the /etc/init.d directory with a file name of **wmsgd**, and symbolic links created in /etc/rc2.d as follows:

```
K80wmsgd -> ../init.d/wmsgd
S80wmsgd -> ../init.d/wmsgd
```

Send feedback to Microsoft