



INTERIX[™] for Microsoft[®] WINDOWSNT[®]

INTERIX Release 2.2 Software Development Kit User's Guide

Softway Systems, Inc.

185 Berry Street, Suite 5514

San Francisco, CA 94107

Tel. (415) 896-0708

Fax (415) 896-0709

<http://www.interix.com>

Softway
Systems

INTERIX Release 2.2 Software Development Kit User's Guide

Written by John McMullen.

The chapter "Programming with Curses" is adapted from the documentation for Ncurses, and was originally written by Eric S. Raymond and Zeyd M. Ben-Halim.

Copyright © 1995-1997, 1998 Softway Systems, Inc. All rights reserved. You may not copy this document in whole or in part without prior written permission from Softway Systems, Inc. Softway Systems makes no representations or warranties other than the standard warranty with regard to the contents of this manual or to the suitability of this software product for any particular purpose.

Softway Systems reserves the right to make any changes to this software product and to revise the information about the products contained in this guide without an obligation to notify any persons about such revisions or changes.

Portions of this software distribution are derived from code owned by the Regents of the University of California. Copyright © 1989, 1993 The Regents of the University of California. All rights reserved.

Portions of this software distribution are derived from Free Software Foundation, Inc. source code. As required by the FSF license, changes to the source code are available from the OPENNT web site's Tool Warehouse. The URL is <http://www.opennt.com/>.

Motif carries the following copyrights: © 1989, 1990, 1991, 1992 Open Software Foundation, Inc., © 1987, 1988, 1989, 1990, 1991, 1992 Hewlett-Packard Company, © 1987, 1988, 1989, 1990 Digital Equipment Corporation, Maynard, Mass., © 1988 Massachusetts Institute of Technology, © 1988 Microsoft Corporation, © 1990 Motorola Inc., © 1989 The Regents of the University of California, © 1990 Conor P. Cahill, © 1991 by Joseph Friedman, © 1988, 1989 by Patrick J. Naughton. All rights reserved.

Trademarks

INTERIX is a trademark of Softway Systems, Inc.

Microsoft, Windows NT, and Microsoft Visual C/C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions.

Pentium and Pentium Pro are registered trademarks of Intel Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through The Open Group.

Open Software Foundation, OSF, OSF/Motif, and Motif are trademarks of The Open Group.

DEC, Digital Alpha, and Digital are registered trademarks of Digital Equipment Corporation

X Window System is a trademark of X Consortium, Inc.

All other products mentioned are trademarks or registered trademarks of their respective companies.

Contents

Introduction	7
Welcome	7
Software Requirements	7
OPENNT to INTERIX	8
Microsoft Compilers	8
Spaces In Directory Names	8
Installing Microsoft Visual C/C++	9
Compatibility Between INTERIX Releases	10
Limitations on the Demo SDK	10
Contacting Technical Support	11
The Software Development Kit	13
The Utilities	13
The Header Files	15
The Libraries	15
The Build Environment	17
The Directory Structure	17
GCC and G++	18
Object File Compatibility	19
Using cc and c89	19
Finding Libraries and Header Files	21
Defined Symbols	21
Isolating INTERIX-Specific Features	22
Make	22
Converting Makefiles	22
POSIX-Conformant Makefiles	22
Porting Code	25
System Overview	25
Writing POSIX-Standard Code	26
Compile-Time Macros and Manifest Constants	27
System Information	27
Advisory File Locking	28
Current Working Directory	28
Error Messages	28
The gets() Function	28
Mknod()	28
POSIX Terminal I/O	29
Process Environment	30
Regular Expression Code	30

Signals	30
Stdarg Vs. Varargs	34
Temporary Files	35
Utime() Vs. Utimes()	35
Extensions to POSIX.1	36
Include Files	36
Memory Mapped Files	36
System V IPC Mechanisms	37
Sockets.....	37
BSD String and Memory Functions	37
Curses and Terminal Routines on INTERIX	37
Pseudoterminals	38
Controlling Terminals	38
INTERIX-Specific Issues	38
User Authentication	38
Absolute Pathnames	41
Further Reading	42
Programming with Curses	43
Introduction	43
Scope of This Document	44
Terminology	44
An Overview of Curses	45
Compiling Programs Using Curses	45
Updating the Screen	45
Standard Windows and Function Naming Conventions	46
Variables	47
Using the Curses Library	47
Starting Up	49
Output	49
Input	50
Using Forms Characters	50
Character Attributes and Color	51
Finishing Up	51
Function Descriptions	52
Initialization and Wrapup	52
Causing Output to the Terminal	53
Low-Level Capability Access	53
Debugging	54
Hints, Tips, and Tricks	55
Some Notes of Caution	55
Temporarily Leaving Ncurses Mode	55
Handling Multiple Terminal Screens	56
Testing for Terminal Capabilities	56
Tuning for Speed	57

Special Features of Ncurses	57
Compatibility with Older Versions	57
Refresh of Overlapping Windows	57
XSI Curses Conformance	58
Further Reading	59
Services	61
The Service Interface	61
Service Execution Environment	62
Coding Service Applications	63
Building X Window Applications	65
The X Directory Hierarchy	65
Utilities	66
Libraries and Header Files	66
Building an Application	67
The INTERIX Motif Development Kit	68
System Error Codes	69
Index	77

WELCOME

SOFTWARE REQUIREMENTS

OPENNT TO INTERIX

MICROSOFT COMPILERS

COMPATIBILITY BETWEEN INTERIX RELEASES

LIMITATIONS ON THE DEMO SDK

CONTACTING TECHNICAL SUPPORT

CHAPTER 1

Introduction

Welcome

Thank you for purchasing the INTERIX Software Development Kit. This User Guide provides an introduction to compiling and porting software in the INTERIX environment. It also describes the features of the INTERIX Motif Development Kit.

Software Requirements

For installation instructions, see the INTERIX *Release 2.2 Getting Started* booklet.

Before installing the INTERIX Software Development Kit, you must have installed the following software:

- Microsoft Windows NT 3.51 or above
- INTERIX Workstation Lite, Workstation, or Server, Release 2.2

Before you install the INTERIX Motif Development Kit, you must have installed the INTERIX Software Development Kit.

OPENNT to INTERIX

Earlier releases of INTERIX before release 2.2 went under the name OPENNT. To preserve backwards compatibility throughout the 2.x releases, the OPENNT pathnames, environment variables, and compiler macros are still valid. The name of the installation directory is still available through the environment variable *OPENNT_ROOT*.

Eventually, future releases will migrate to new names (such as renaming *OPENNT_ROOT* to be *INTERIX_ROOT*), but the old ones will remain for backwards compatibility during the 2.x series of releases. For example, if you are installing the product over a previous release, the old installation directory is used. If this is a fresh installation, the directory **C:\Interix** is used. Eventually, occurrences of “OpenNT” will be replaced by “Interix”.

Microsoft Compilers

The INTERIX Software Development Kit supports the use of Microsoft Visual C/C++ (MSVC), version 4.0 or higher, through the *cc* and *c89* compiler drivers. The *cc* and *c89* utilities are interfaces to the compiler and linker programs in MSVC (*CL.EXE* and *LINK.EXE* respectively).

The *cc* and *c89* programs work only with the MSVC compiler, not with *gcc*.

This section describes how the *cc* and *c89* programs find the MSVC compiler and linker. There are two issues:

- Handling spaces in directory names. The default installation location for MSVC is in the directory **Program Files**; many utilities (such as *makedepend*) do not handle spaces in directory names well, if at all. You may need to worry about this even if you select support for Microsoft compilers.
- Installing Microsoft Visual C/C++ *after* installing the INTERIX SDK. In this case, you must point the *cc* and *c89* programs to the MSVC compiler and linker.

Spaces In Directory Names

The default install path for MSVC includes the directory **Program Files** (including the space). If there are spaces in the directory name, there

are spaces in the value of the *PATH*. The most common symptom is that the shell cannot find *cc* or *c89*.

Change any pathname component which includes spaces to the 8.3 name. (Windows NT stores both long filenames and 8.3 versions of the filenames.) To display the 8.3 name of a given file or directory, use the File Manager to display the appropriate directory. Open the **View** menu and select the item *All File Details*. File Manager displays the 8.3 name for all long file names in the directory.

For example, if the *PATH* contains the directory name **Program Files** which has the 8.3 name **PROGRA~1**, replace the string “Program Files” in your *PATH* with “PROGRA~1”.

Your path may look something like this (line broken to fit):

```
PATH=$OPENNT_ROOT/bin:$OPENNT_ROOT/contrib:\
//C/PROGRA~1/MSDEV/BIN://C/PROGRA~1/MSDEV/IDE
```

You may want to make the same change to the value of the system environment variable *OPENNT_COMPILERDIR*, which is described in the next section.

Installing Microsoft Visual C/C++

If MSVC is already installed when you install the INTERIX SDK and you select support for Microsoft Compilers, you do not need to worry about this.

If you install MSVC *after* installing the INTERIX SDK, you must point *cc* and *c89* at the compiler and linker. The INTERIX shell startup files (*\$OPENNT_ROOT/etc/profile* for the Korn Shell and *csh.login* for the C Shell) contain code to automatically add the appropriate directories to your *PATH*, based on the value of the Windows NT environment variable *OPENNT_COMPILERDIR*.

This environment variable must be set using the **System** applet of the Control Panel. Setting it in your INTERIX environment will have no effect.

- 1 Install Microsoft Visual C/C++. **Ensure the install path contains no spaces.** If the install path contains spaces, you will have to follow the procedure outlined in the previous section, “Spaces In Directory Names.”
- 2 Open the **System** applet in the Control Panel.

- 3 Add the environment variable *OPENNT_COMPILERDIR*. The value of the variable is the directory in which you installed MSVC, in the POSIX format. For example, if you installed MSVC in the directory **C:\MSDEV**, the value of *OPENNT_COMPILERDIR* should be **//C/MSDEV**.
- 4 Save the change and exit.

Compatibility Between INTERIX Releases

A program built using OPENNT SDK Release 2.0 or 2.1 should still run on an INTERIX Release 2.2 system. Executables still run on later releases of INTERIX with the same major release number, but may not run on earlier releases. That is, a program built using INTERIX SDK Release 2.2 may not run on an OPENNT Release 2.1 system.

Because of the OPENNT to INTERIX name change, items such as registry variables will change names in Release 3.0. Plan now to recompile applications when the major release number changes.

Limitations on the Demo SDK

The demonstration version of the INTERIX Software Development Kit is fully functional, but the executables built using the demonstration version have one limitation:

Executables built using the demonstration version run only on an INTERIX system which has the demonstration Software Development Kit installed. They will not run on an INTERIX Workstation Lite, Workstation, or Server system unless the demo SDK has been installed.

If the demonstration period on an INTERIX Workstation Lite, Workstation, or Server system has expired, the executable will not run. To remove this limitation, rebuild the executable using a licensed version of the INTERIX Software Development Kit.

Contacting Technical Support

Softway Systems, Inc. provides 30 days of free technical support with every product, beginning with first contact. Support beyond the thirty-day period is available on a contract basis. Please contact sales@interix.com for more information.

Technical support is provided between the hours of 9 a.m. and 5 p.m. Pacific Standard Time. E-mail is the best method for contacting technical support.

Please include your name, a phone number where you can be reached, and a return e-mail address if it is not the same as that from which you sent the e-mail. Your problem description should include the INTERIX version number, the installed components, and your registered serial number.

You can reach Softway Systems by any of the following:

E-mail: support@interix.com

Phone: (519) 571-7104

Fax: (415) 896-0709

FTP server: <ftp.interix.com> (use anonymous login)

Web Site: <http://www.interix.com/>

Mail: Softway Systems, Inc.
Technical Support
185 Berry Street, Suite 5514
San Francisco, CA 94107
USA

THE UTILITIES

THE HEADER FILES

THE LIBRARIES

OTHER DOCUMENTATION

CHAPTER 2

The Software Development Kit

This chapter describes the components of the INTERIX Software Development Kit. The next chapter describes the build environment and tools.

The Utilities

The Software Development Kit provides the following utilities:

<i>ar</i>	Archive files and manage libraries.
<i>as</i>	Generate object code from assembler.
<i>c89</i>	POSIX.2-conforming compiler front-end. Invokes the Microsoft Visual C/C++ <i>CL.EXE</i> compiler.
<i>cc</i>	Traditional-style compiler front-end. Invokes the Microsoft Visual C/C++ <i>CL.EXE</i> compiler.
<i>ci</i>	Place a file under revision control or add new revision (check-in).
<i>co</i>	Check out working copy of a file under revision control.

<i>cpp</i>	C preprocessor. This is not used by <i>c89</i> (the Microsoft compiler includes its own preprocessor), but is used by some of the other tools, such as <i>gcc</i> and <i>imake</i> .
<i>g++</i>	Front end for GNU C++ compiler.
<i>gcc</i>	Front end for GNU C compiler. Also compiles C++ code.
<i>ident</i>	Identify revision of a file included in a binary file.
<i>imake</i>	Creating makefiles from template files using C preprocessor.
<i>ld</i>	Linker; normally never called directly (use <i>gcc</i> or <i>g++</i>).
<i>lex</i>	Generate lexical analyzer code.
<i>makedepend</i>	Create dependency list from source files.
<i>make</i>	Keep generated files up-to-date with respect to their source files.
<i>mkdirhier</i>	Create directory hierarchies; equivalent to mkdir -p .
<i>nm</i>	List symbolic names in an object file, library or executable.
<i>rcs</i>	Manipulate files under revision control.
<i>rcsdiff</i>	Display differences between revisions of a file under revision control.
<i>rlog</i>	Display change history of a file under revision control.
<i>strerror</i>	Display the error string associated with an error return code.
<i>strip</i>	Remove symbol table from an object file.
<i>yacc</i>	Generate LALR parser code from a grammar description.
<i>xmkmf</i>	Create a makefile from an Imakefile .

Reference pages for these utilities can be found using the *man* command and in the *INTERIX Commands and Utilities Reference*.

Gcc and *g++* are ports of the popular GNU C compiler (version 2.7).

The *make* utility in this release is based on the BSD 4.4 *make*. The *lex* and *yacc* utilities are based on *flex* and BSD *yacc*. The RCS utilities are based on the RCS 5.6 utilities.

Strerror is a useful information utility which displays the error string

associated with a particular return code. To see the error string for an error *n*, give the command

```
strerror n
```

To see the full list of error strings, use the **-l** (letter ell) option:

```
strerror -l
```

The Header Files

Most of the header files supplied with the INTERIX Software Development Kit are described in the relevant standards.

If you are using a Microsoft compiler, the Microsoft header files are never included; the INTERIX header files are complete in themselves.

The Libraries

The INTERIX Software Development Kit contains the POSIX.1 interfaces, the POSIX.2 interfaces, the ISO / ANSI C libraries, a wealth of interfaces and libraries found on open systems, and libraries for *lex*, *yacc*, *curses*, and *termcap*. It also provides the C runtime startup code and other useful functions. These interfaces and libraries are gathered into the following libraries:

libc.a	INTERIX functions and replacements for some standard C library calls. These libraries also contain the math routines.
libcurses.a	Curses library routines.
libl.a	<i>Lex</i> library; normally replaced by a custom-written function.
libm.a	Math routines library.
libsocket.a	This library is a dummy library, provided so Make-files don't have to be changed; it isn't required to link with the socket calls.
libtermcap.a	Terminal interface functions. Both <i>terminfo</i> and <i>termcap</i> functions are supported.
liby.a	<i>Yacc</i> library; normally replaced by a custom-written function.

psxdll.a Symbols for functions in the INTERIX subsystem DLL.

The GCC/G++ libraries are installed in **\$OPENNT_ROOT/usr/contrib/machine/cygnus-2.7.2-970404**. On a DEC Alpha system, the value of *machine* is **alpha-pc-opennt**; on an Intel-architecture machine, the value of *machine* is **i386-pc-opennt**. The libraries are:

libgcc.a GNU C++ library. Terms and conditions of use are in the file **\$OPENNT_ROOT/pubs/GLPL.TXT**.

libg++.a GNU C++ library. Terms and conditions of use are in the file **\$OPENNT_ROOT/pubs/GLPL.TXT**.

libstdc++.a GNU standard C++ library. Terms and conditions of use are in the file **\$OPENNT_ROOT/pubs/GLPL.TXT**.

The X11 libraries (installed in **\$OPENNT_ROOT/usr/X11R5/lib**) are:

libX11.a Xlib routines.

libXau.a Xauthority routines library.

libXaw.a Athena Widget library

libXdmcp.a XDM control protocol library.

libXext.a X11 extensions library.

libXi.a X11 intrinsics library.

libXmu.a Miscellaneous utilities library.

libXt.a X11 toolkit library.

libXtst.a Test routines.

liboldX.a X10 compatibility library.

THE DIRECTORY STRUCTURE

GCC AND G++

USING CC AND C89

MAKE

CHAPTER 3

The Build Environment

This chapter discusses building software with the INTERIX Software Development Kit. The INTERIX SDK provides the compiler interfaces *cc*, *c89*, *gcc*, and *g++*. To use the *cc* and *c89* compiler interfaces, you must already have Microsoft Visual C/C++ set up and running in your environment. The header files provided work with both MSVC and *gcc* compilers.

The *cc* and *c89* interfaces will not allow you to compile C++ code. You must use *g++* for C++ code.

The Directory Structure

When you installed the INTERIX SDK, it created the following directory hierarchy relative to the INTERIX install directory; the name of that directory is stored in the *OPENNT_ROOT* environment variable.

```
usr/include  
usr/include/arpa  
usr/include/g++  
usr/include/gnu  
usr/include/netinet  
usr/include/opennt  
usr/include/sys
```

```
usr/lib
usr/share/man/cat2
usr/share/man/cat3
usr/share/man/cat5
usr/X11R5/include
usr/X11R5/include/X11
usr/X11R5/lib
usr/X11R5/lib/X11
usr/X11R5/man/cat3
```

The utilities provided with the INTERIX Software Development Kit are installed in **\$OPENNT_ROOT/bin**, **\$OPENNT_ROOT/usr/X11R5/bin**, **\$OPENNT_ROOT/usr/contrib**, and in **\$OPENNT_ROOT/usr/contrib/bin**.

GCC and G++

The INTERIX SDK also includes the *gcc* and *g++* compiler, front ends, and the *libg++* C++ library. The *gcc/g++* compiler doesn't require Microsoft Visual C/C++ to run, and it doesn't prevent the use of Microsoft Visual C/C++.

Note: The *gcc/g++* compiler, front ends and the **libg++** library are distributed under the terms of the GNU license agreements. Be sure you understand the Free Software Foundation's General Public License before using *gcc* or *g++*, and be sure you understand the Library General Public License before linking with the **libg++** C++ library. The license agreements are in the files **GPL.TXT** and **LGPL.TXT** in the **\$OPENNT_ROOT/pubs** directory.

The *gcc/g++* package includes a number of binaries installed in **\$OPENNT_ROOT/usr/contrib/bin**, including *as*, *g++*, *gcc*, *gcov*, *ld*, *objdump*, and *protoize*. Other files are installed in one of the directories **\$OPENNT_ROOT/usr/contrib/alpha-pc-opennt/cygnus-2.7.2-970404** or **\$OPENNT_ROOT/usr/contrib/i386-pc-opennt/cygnus-2.7.2-970404**, for DEC Alpha or X86 machines, respectively.

The *cc/c89* utilities will not recognize *gcc*; to use *gcc* or *g++*, you must modify your **Makefiles** so that **CC=gcc** or **CC=g++**.

Object File Compatibility

The *gcc* linker can link the object files created by either *cc/c89* or by *gcc/g++* without problems. The MSVC linker used by *cc/c89* will not usually be able to link object files created by *gcc/g++*. Although the object files created by *gcc/g++* are in a compatible format, they contain additional symbol types and conventions.

Using *cc* and *c89*

The INTERIX Software Development Kit provides standard compiler interfaces, *cc* and the POSIX interface utility, *c89*. (They are actually the same utility, so references to *cc* apply to *c89* unless otherwise mentioned.) To build a program with *c89*, run it as you would *cc*. For example:

```
c89 -o hello hello.c
```

Unlike *cc*, the order of operands is important to *c89*: most options must come before positional operands. You cannot say:

```
c89 io.o main.o -o program
```

Instead, you must say:

```
c89 -o program io.o main.o
```

This doesn't apply to **-l** library options, however; those are placed at the end of the command line in the order they are to be searched:

```
c89 -o program io.o main.o -lmylibc -ltermcap
```

Both *cc* and *c89* recognize the most common compiler options:

- c** Compile but do not link/edit, and don't remove object files.
- D *name*[=*value*]** Define *name* as if **#define** had been used. If *value* is not given, *name* is defined as 1.
- E** Expand all C preprocessor directives and copy the C source to standard output.
- g** Include symbol information in the object or executable file.

- I *directory*** Search *directory* for include files before searching the usual places (normally **//C/Interix/usr/include**).
- L *directory*** Search *directory* for libraries before searching the usual places (normally **//C/Interix/usr/lib**).
- M** Produce a linker map file. This option only has an effect if the linker is actually invoked.
- N [**nostdlib** | **nostdinc** | **nostdc**]**
Disables standard C features; can be given multiple times. **nostdlib** prevents the inclusion of standard library files; **nostdinc** prevents the inclusion of standard header files; **nostdc** turns on the Microsoft compiler extensions, and undefines the **__STDC__** macro. The **-N nostdc** behavior is the default for *cc* but not for *c89*.
- O** Optimize the generated code.
- o *outfile*** Name the output executable file *outfile*. This option cannot be used with **-c**.
- P** Preprocess the code and place in a file with the extension **.i**.
- S** Create an assembler listing; the format and the file name depend upon the underlying compiler.
- s** Strip all nonessential information from the object file or executable.
- U *name*** “Undefine” *name* as if the **#undefine** keyword had been given.
- X *CLopt*** Pass the option string *CLopt* directly to the **CL.EXE** utility.
- Y *Linkopt*** Pass the option string *Linkopt* directly to the **LINK.EXE** utility.

Cc and *c89* invoke the Microsoft Visual C/C++ utilities **CL.EXE** (for compiling) and **LINK.EXE** (for linking). They search your *PATH* for the programs. If you don’t want to specify the Microsoft Visual C/C++ binaries directory in your *PATH*, you can also specify the full pathnames of the compiler and the linker with the environment variables **C89_COMPILER** and **C89_LINKER**. For example:

```
export C89_COMPILER=//C/MSVC40/BIN/CL.EXE
export C89_LINKER=//C/MSVC40/BIN/LINK.EXE
```

This will not work for Microsoft Visual C/C++ 5.0, because it searches the *PATH* for DLLs.

If you need to see the command line being passed to *CL.EXE*, you can set the environment variable *C89_ECHO*. When this is set to 1, *c89* echoes command lines before executing them.

Finding Libraries and Header Files

Microsoft linkers normally use the *LIB* environment variable to specify alternate search locations for libraries. To avoid conflicts with Windows utilities, the INTERIX *cc* and *c89* utilities ignore the initial value of *LIB*. To add additional libraries to the search path, use the **-L** option. You can specify it multiple times on the command line.

You can also specify libraries with the traditional operands **-lc**, **-ll**, **-lm**, and **-ly**. These specify the C library, the *lex* library, the math library and the *yacc* library, respectively.

Many compilers make use of the *INCLUDE* environment variable to specify alternate search locations for header files. To avoid conflict with Windows utilities, the *c89* utility ignores the initial value of *INCLUDE*. To add additional directories to the search path, use the **-I** option. You can specify it multiple times on the command line.

Defined Symbols

The *c89* utility defines the following macro symbols to be 1: `_POSIX_` and `__OPENNT`. The `_POSIX_` symbol is a reserved symbol, and shouldn't be used or modified. The *cc* utility defines the symbols `__OPENNT` and `unix`. (The "unix" macro is defined because many application sources intended to compile on multiple platforms use this simple macro to call out features found on UNIX systems, as opposed to DOS or VMS.)

C89 by default passes the `/Za` option to the Microsoft compiler, which defines `__STDC__`, unless you specify **-N nostdc**. The *cc* utility by default passes the `/Ze` option and the Microsoft compiler does not define `__STDC__`.

Isolating INTERIX-Specific Features

If you want to add or isolate features specific to INTERIX, you can surround it with an `#ifdef __OPENNT`. This macro is automatically defined by `cc` and `c89`.

Make

The version of *make* provided conforms to POSIX.2 and is based on the BSD 4.4 *make*. It supports all BSD features.

Converting Makefiles

To convert existing **Makefiles** to run under INTERIX, you will have to change macro definitions:

- If your **Makefiles** define macros such as BINDIR, INCDIR and LIBDIR, prefix the values with the INTERIX install directory, **\$(OPENNT_ROOT)**.
- Edit the value of **\$(CFLAGS)** to ensure that only flags supported by `cc` or `c89` are used.

POSIX-Conformant Makefiles

If your **makefiles** are required to be strictly POSIX-conformant, you must do two things:

- Include the target .POSIX before any other non-comment or non-blank lines in the makefile; and
- Change your **Makefiles** so that **\$(CC)** is defined to be `c89`.
- Make sure that **\$(CFLAGS)** contains only `c89` flags.
- Avoid using any features of *make* not mentioned in the POSIX.2 standard.

The extensions to POSIX.2 provided in the INTERIX *make* are:

- The special targets `.BEGIN`, `.END`, `.INTERRUPT`, `.MAIN`, `.NOTMAIN`, `.MAKE`, `.MAKEFLAGS`, `.OPTIONAL`, `.PATH`, and `.USE`.
- The special macros `.TARGET`, `.OODATE`, `.PREFIX`, `.MEMBER`, `.IMPSRC`, `.ALLSRC`, and `.ARCHIVE`.
- An include directive, **`.include`**, for including other files (the **`-I`** option specifies the directories to search for those other files).
- Conditionals such as **`.if`**, **`.ifdef`**, **`.ifmake`**, and so on.

See the *make* reference page for more details on these extensions.

OVERVIEW

SYSTEM OVERVIEW

MAKING CODE PORTABLE

EXTENSIONS TO POSIX.1

INTERIX-SPECIFIC CHANGES

CHAPTER 4

Porting Code

This chapter discusses the strategies for porting code from other operating systems to INTERIX. Information on the INTERIX implementation of specific packages (such as *curses* or the X Window System) is in later chapters.

If you're porting a traditional daemon program to INTERIX, you should also read chapter 5, "Services".

System Overview

The INTERIX subsystem is a POSIX.1-conformant subsystem. It supports sockets, BSD 4.4 interfaces and the System V IPC mechanisms, pseudo terminals, memory mapped files, and a wealth of other functionality.

Applications are often written for specific operating systems and then ported to others, using some combination of conditional compilation (**#ifdef** preprocessor commands) or wrapper functions. Porting such software to INTERIX has two parts:

- Making the source portable. This is choosing the features supported by INTERIX: the POSIX.1 interface and some extensions taken from other standards (such as the *Single UNIX Specification*) and from traditional systems such as BSD or System V derivatives.
- Dealing with issues specific to INTERIX.

The typical porting process is to type *make* in the source directory and then fix the problems as the compiler reports them. The most common changes required are:

- Select the appropriate compile flags for a POSIX.1 system.
- Replace the `varargs` macros with `stdarg` macros.
- Replace references to absolute pathnames (such as `/bin/sh`) with INTERIX-style pathnames, either by using `confstr()` or by using the INTERIX routines supplied.
- Isolate symbolic links with `#ifdef S_IFLNK`. When symbolic links are implemented, this macro will be defined in `<sys/stat.h>` and the code will be included at the next compilation.
- Isolate the **struct stat** members `st_rdev`, `st_blksize`, and `st_blocks`.

A set of ported sources are installed in `$OPENNT_ROOT/usr/examples`. For each file which required changes from the publicly-available source, a diff file has been included.

Writing POSIX-Standard Code

The POSIX standard (1990) replaced the generic types in the APIs (such as **int**) with more specific types (such as **pid_t** or **mode_t**). Normally you can fix these problems in your code by checking the prototype in the headers and changing the variable declaration.

This table summarizes the choices made for portable code:

Historical	INTERIX (POSIX.1)
MAXPATHLEN	PATH_MAX
MAXUPRC	CHILD_MAX
NCARGS	ARG_MAX
NGROUPS	NGROUPS_MAX
NOFILE	OPEN_MAX
<code>getwd()</code>	<code>getcwd()</code>
<code>index()*</code>	<code>strchr()</code>
<code>lockf()</code>	<code>fcntl()</code>
<code>mknod()</code>	<code>mkfifo()</code>
<code>mkstemp()*</code>	<code>tmpfile()</code>
<code>mktemp()*</code>	<code>tmpnam()</code> or <code>tempnam()</code>
<code>rindex()*</code>	<code>strrchr()</code>
<code>signal()*</code>	<code>sigaction()</code>
<code>utimes()</code>	<code>utime()</code>

Interfaces marked with * are also supplied in INTERIX.

The APIs *mkstemp()*, *mktemp()*, *index()*, *rindex()*, and *signal()* are defined in the *Single UNIX Specification*, and are supplied in INTERIX. The APIs *lockf()* and *getwd()* are defined in the *Single UNIX Specification* but are not yet supplied in INTERIX.

Compile-Time Macros and Manifest Constants

POSIX specifies its own set of compile-time macros and manifest constants, not defined in either System V or BSD systems.

Programs written in a BSD environment often include **<sys/param.h>**. The INTERIX SDK provides this header file—it includes **<sys/types.h>** and **<limits.h>**.

You may have to change manifest constants in your code, as described in the previous section.

System Information

There are different historical ways to determine information about the system. On BSD systems, the *sysctl()* interface provides access to system information, while on System V systems, the *sysinfo()* call provides system information.

Some of this information can be obtained using POSIX routines and some cannot. The POSIX system information routines return strings, paths, and numeric values (including two-valued boolean conditions). The header file **<limits.h>** also contains macros defining system limits.

confstr() Retrieve string values; the only portable value is *_CS_PATH*, a value for *PATH* guaranteed to find the standard utilities.

fpathconf(), *pathconf()* Retrieve configurable pathname variables such as the maximum size of a file name or the maximum link count. Familiar to System V programmers.

sysconf() Retrieve system information, such as whether job control is available, the POSIX options supported, limits for *bc*, and the maximum number of bytes allowed as an argument to *exec()*.

INTERIX also provides both *uname()* and *gethostid()*. See the reference pages for more information on all of the functions provided.

Advisory File Locking

The INTERIX subsystem supports advisory file locking through the *fcntl()* routine, with the commands *F_GETLK*, *F_SETLK*, and *F_SETLKW*. The System V *lockf()* and BSD *flock()* interfaces are not supported; if you need *lockf()*, it's possible to write an implementation built on *fcntl()*.

Current Working Directory

Some code still exists that uses *getwd()* to determine the current working directory; INTERIX provides the POSIX *getcwd()* interface instead.

Note that *getcwd()* does not support being called with a null pointer.

Error Messages

The system error messages are available through the *strerror()* and *perror()* calls. Traditional systems expose the underlying *sys_errlist* array and store the number of elements in the array in the variable *sys_nerr*. INTERIX provides the calls but does not expose the arrays.

The gets() Function

The *gets()* function is a known security hole, and your applications shouldn't use it. Use *fgets()* instead.

This implementation prints out a warning whenever the *gets()* function is called: "warning: this program uses gets(), which is unsafe." To disable the warning, set the environment variable *DISABLE_GETS_WARNING*.

Mknod()

Use *mkfifo()* rather than *mknod()*. The only portable purpose of *mknod()* is for creating FIFO-special files, and POSIX.1 already pro-

vides the *mkfifo()* call for that. Other uses of *mknod()*, such as creating other device special files, aren't currently supported under INTERIX.

POSIX Terminal I/O

There have traditionally been two terminal hardware interfaces: the System V **termio** structures and the BSD **stty** structures, generally controlled with *ioctl()* calls. POSIX replaced both of these with the **termios** structure and a new set of access calls.

The I/O model is very similar to the System V model. There are two modes, canonical and non-canonical. Canonical input is line-based (the same as BSD cooked mode); non-canonical mode is character-based (like BSD raw or cbreak mode). As of release 2.0, the INTERIX extended POSIX subsystem includes a true non-canonical mode, with support for *cc_c[VMIN]* and *cc_c[VTIME]*.

The **termios** structure is defined in **<termios.h>**:

```
struct termios {
    tcflag_t c_iflag;    /* input flags */
    tcflag_t c_oflag;    /* output flags */
    tcflag_t c_cflag;    /* control flags */
    tcflag_t c_lflag;    /* local flags */
    cc_t      c_cc[NCCS]; /* control characters */
};
```

INTERIX extends the POSIX.1 set of flags for **c_iflag** to include IMAXBEL and VBELTIME. For **c_cc**, note that VMIN and VTIME do *not* have the same values as VEOF and VEOL, though a portable application should take into account that they are allowed to be identical on a POSIX.1 system.

The following new functions replace the terminal I/O *ioctl()* calls (such as *ioctl(fd, TIOCSETP, buf)* and *ioctl(fd, TIOCGETP, buf)* or *stty()* and *gtty()*). They were changed because the data type of the final argument for terminal I/O *ioctl()* calls depends on the action; that makes type checking impossible. Instead, these twelve functions are defined:

<i>tcgetattr()</i>	fetch attributes (termios structure)
<i>tcsetattr()</i>	set attributes (termios structure)
<i>cfgetispeed()</i>	get input speed
<i>cfgetospeed()</i>	get output speed
<i>cfsetispeed()</i>	set input speed
<i>cfsetospeed()</i>	set output speed

<i>tcdrain()</i>	wait for all output to be transmitted
<i>tcflow()</i>	suspend transmit or receive
<i>tcflush()</i>	flush pending I/O
<i>tcsendbreak()</i>	send BREAK character
<i>tcgetpgrp()</i>	get foreground process group ID
<i>tcsetpgrp()</i>	set foreground process group ID

The `TIOCGWINSZ` command for `ioctl( )` and the **winsize** structure are supported, if you need to get the window size.

Process Environment

If you're porting SVID code, you may need to be careful of the process environment. Rather than defining the process environment as a third argument to `main( )`, it's 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, you can use `getenv( )`, `setenv( )`, and `putenv( )`, or you can build a new environment and pass it to the child using the `envp` argument of `exec( )`.

Regular Expression Code

INTERIX supports the POSIX regular expression functions `regcomp( )`, `regex( )` and `regerror( )`. They are declared in **<regex.h>**.

Signals

INTERIX supports three different sets of signal-handling APIs, although it only supports one set of signal semantics (the POSIX.1 set):

- ANSI C signals, supported with the function `signal( )`; because this API is built on top of the POSIX.1 `sigaction( )` model, it behaves slightly differently than you might expect
- POSIX.1 signals, supported with the functions `sigaction( )`, `sigpending( )`, `sigprocmask( )`, `sigsuspend( )`, `sigemptyset( )`, `sigfillset( )`, `sigaddset( )`, `sigdelset( )`, `sigismember( )`.
- BSD 4.3 signals are supported with the functions `killpg( )`, `sigsetmask( )`, `sigblock( )`, and `sigvec( )`. The signal mask for these functions is an **int**, not a **sigset_t**. If a future release of INTERIX

supports more than 32 signals, these functions will become obsolete. Rather than depend on them, convert your code to use the POSIX.1 signal calls. Although *sigpause()* is provided, it is provided as the System V call, which does not behave in the same way as the BSD call.

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

There are two important things to remember about signals on INTERIX:

- Signals on a POSIX.1 system are neither BSD nor SVID. POSIX defined a new signal mechanism based on the API *sigaction()*.
- The set of signals available is the signals described in POSIX.1 and the Single Unix Specification. Your code should use the symbolic names instead of the actual values, since some numbers may differ from traditional implementations.

The POSIX.1 committee introduced the new signal semantics because of problems with historical signal implementations found on BSD and System V systems. When the System V3 *signal()* catches a signal, the action associated with the signal is reset to the default. In 4.3BSD, it is not reset. In the ISO / ANSI C standard, the *signal()* function either resets the default or does an implementation-defined blocking of the signal. The POSIX *sigaction()* call does not reset the default if the handler returns normally. INTERIX follows the POSIX signal semantics.

An INTERIX process has a signal mask: it can choose to block certain signals from arriving. (You cannot block SIGKILL or SIGQUIT.) A process starts with a signal mask inherited from its parent. If any signals are generated and then blocked by the signal mask, they go into the set of pending signals.

If you're using the *signal()* API, note that the signal is still masked and remains masked until the mask is cleared. This can be a real problem if your code does a *longjmp()* from the handler. Converting to use *sigaction()* directly and *siglongjmp()* will clear up some of those unexpected behaviors.

Available Signals

The following signals are generated on INTERIX. The default action for all of them is termination of the process. (Other signals are defined in **<signal.h>**, so you can pass them to *kill()*, but are not generated.)

Signal	Meaning
--------	---------

SIGABRT	Abnormal termination (see <i>abort()</i>)
SIGALRM	Timeout (see <i>alarm()</i>)
SIGFPE	Erroneous arithmetic operation
SIGHUP	Controlling terminal hung up
SIGILL	Hardware interrupt (illegal instruction)
SIGINT	Interactive interrupt
SIGIO	I/O is possible on a descriptor
SIGKILL	Termination (cannot be caught or ignored)
SIGPIPE	Write on a pipe that isn't open for reading by any process
SIGQUIT	Interactive termination signal
SIGSEGV	Invalid memory reference
SIGTERM	Termination signal
SIGURG	Urgent condition on socket
SIGUSR1	Application-defined signal 1
SIGUSR2	Application-defined signal 2
SIGVTALRM	Virtual time alarm
SIGWINCH	Window size change

The following job-control signals are provided:

Signal	Meaning	Default Action
SIGCHLD	Child process stopped or terminated	Ignored
SIGCONT	Continue	Continue if stopped
SIGSTOP	Stop (cannot be caught or ignored)	Stop process
SIGTSTP	Interactive stop	Stop process
SIGTTIN	Read from controlling terminal by background process	Stop process
SIGTTOU	Write to controlling terminal by background process	Stop process

Note that the SIGCHLD signal is not the System V SIGCLD signal. It is similar but has slightly different semantics.

As of INTERIX 2.1, SIGCLD is synonymous with SIGCHLD, and SIGIOT is synonymous with SIGABRT.

Signal Data Structures

Information regarding signals in POSIX systems is stored in two data structures: the *signal set* (a **sigset_t**), which represents some subset of

all possible signals, and the *signal action* (represented by a **struct sigaction**), which is a handler function and an additional mask.

```
struct sigaction {
    void      (*sa_handler)( );
    sigset_t  sa_mask;
    int       sa_flags;
};
```

The *sa_handler* member corresponds to the second argument to the *signal()* function. The *sa_mask* adds to the existing signal mask for the process.

The *sa_mask* set of signals are signals to be blocked during the execution of *sa_handler()*. (In fact, the set of signals that are blocked during the execution of the handler function are the union of the process' signal mask and *sa_mask*.)

The SA_RESTART flag provided in BSD is *not* provided in INTERIX.

Other Signal Functions

INTERIX provides the following POSIX functions for manipulating the signal set:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

To manipulate the signal mask, use *sigprocmask()*:

```
int sigprocmask(int how, sigset_t *set, sigset_t
*oset);
```

To check if a signal is pending, use *sigpending()*:

```
int sigpending(sigset_t *set);
```

See the INTERIX *System Interface Reference* or any POSIX.1 reference for explanations of the functions.

Stdarg Vs. Varargs

The ISO/ANSI C standard defined a new mechanism for dealing with variable argument lists, `<stdarg.h>`, rather than conflict with the historical routines in `<varargs.h>`. INTERIX only supports `stdarg`. The `varargs` mechanism uses a magic name (**`va_alist`**) for the first argument in the list, while `stdarg` uses the last required argument. That means there must be at least one named parameter.

It's usually easy to translate from `varargs` to `stdarg`, since most functions with variable argument lists really do have a known first argument type. Here's a trivial error-printing function using the `varargs` mechanism:

```
#include <varargs.h>
prnterror( va_alist );

void prnterror (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 function uses a format string as its first argument, so that can easily be the known argument:

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

You still must give the first argument a name even when the routine takes a terminated list and no fixed arguments at all. For example, this 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);
}
```

Temporary Files

The INTERIX libraries provide the traditional `mkstemp()` and `mktemp()` functions. They also provide the standard functions `tempnam()` and `tmpnam()` to create filenames suitable for temporary files, and the `tmpfile()` function to create a temporary file. The `tmpnam()` function returns a pointer to a temporary file name; `tempname()` does the same, but allows you to specify the directory and file name prefix.

Utime() Vs. Utimes()

Use the POSIX.1 `utime()` function rather than the `utimes()` function. The arguments and semantics are slightly different. The prototype for the `utimes()` function is:

```
int utimes(const char *path, const struct
timeval *times);
```

In the `utime()` function, the second argument, `times`, is a `struct utimbuf`:

```
int utime(const char *path, const struct
utimbuf *times);
```

The `utimes()` function will succeed on writeable files but `utime()` with a non-null argument will *not* succeed, unless the user owns the file.

The `utimbuf` structure is defined in `<utime.h>`; it contains the following two members:

```
time_t actime; /* access time */
time_t modtime; /* modification time */
```

Extensions to POSIX.1

The INTERIX subsystem and libraries provide features beyond strict POSIX.1 conformance. These features include memory mapped files, System V IPC mechanisms, the curses library, and BSD-style TCP/IP sockets.

Include Files

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>`, while those which 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 you define the macro `_POSIX_SOURCE` to be 1 *before* the first header file is included, your program is restricted to the POSIX namespace: it will contain only those APIs specified in the POSIX standards. You may find this restrictive.

To get all of the APIs provided with INTERIX, `#define _ALL_SOURCE` as 1 before the first header file is included. For example:

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

Memory Mapped Files

Note that the value returned by `sysconf(_SC_PAGE_SIZE)` or by `sysconf(_SC_PAGESIZE)` is not the actual virtual memory page size used by the system, it is the value used by the `mmap()` call. You can also get this value using the macro `PAGESIZE`, defined in `<limits.h>`. Using the macro is non-portable, but may be more convenient.

System V IPC Mechanisms

INTERIX provides System V interprocess communication mechanisms: shared memory (*shmat()*, *shmctl()*, *shmdt()*, and *shmget()* routines), message queues (*msgctl()*, *msgget()*, *msgrcv()*, and *msgsnd()* routines), and semaphores (*semctl()*, *semget()*, and *semop()* routines).

Sockets

INTERIX implements the Berkeley-style socket interfaces, including *bind()*, *accept()*, and *connect()*. The INTERIX implementation uses the Windows Winsock library to access the network. Thus all administration is still handled using the usual Windows NT tools.

Only the TCP/IP protocol for sockets is supported, and all of the protocols of the underlying Winsock implementation; see your Winsock documentation for details.

BSD String and Memory Functions

String interfaces specified by ANSI/ISO C are in **<string.h>** while those only in the *Single UNIX Specification* are in **<strings.h>**. The **<strings.h>** file contains:

<i>bcmp()</i>	<i>ffs()</i>	<i>strcasecmp()</i>
<i>bcopy()</i>	<i>index()</i>	<i>strdup()</i>
<i>bzero()</i>	<i>rindex()</i>	<i>strncasecmp()</i>

The BSD4.4 routine *strsep()* is also supported.

Curses and Terminal Routines on INTERIX

The INTERIX libraries include **libcurses.a**, a port of the ncurses package, and **libtermcap.a**, termcap routines.

The curses routines make use of the terminfo database, stored at **\$OPENNT_ROOT/usr/share/terminfo**. (Note that the location is different from traditional systems.) To link with the terminfo routines, link with the curses library.

For more information on this implementation, see chapter 6, or see the *Curses* documentation.

Pseudoterminals

INTERIX supports pseudoterminals. Both the BSD (**/dev/ptynn**) and the System V (**/dev/ptmx**) methods are supported for opening the master side of a pty; the System V method is marginally faster because the search for an available master device is handled in the subsystem.

When using **/dev/ptmx**, the slave tty name can be obtained with `ptsname()`.

Currently, INTERIX supports 265 ptys named **/dev/[p-zA-E][0-9a-f]** on the master side. The corresponding slave side names are **/dev/tty[p-zA-E][0-9a-f]**.

BSD-based `ioctl()` calls can be used with the pty master side.

Controlling Terminals

A process without a controlling tty acquires a controlling terminal on the first `open()` to a tty, provided it is a session leader, unless `NOCTTY` is specified in the `open()` call.

INTERIX-Specific Issues

If your source is already POSIX.1 conformant, you can move directly to dealing with INTERIX issues. There are really only two: user authentication and absolute pathnames. This Software Development Kit provides methods for dealing with each issue. The idiom you choose depends upon your needs.

User Authentication

As mentioned in the *INTERIX Commands and Utilities User's Guide*, the security system on Windows NT implements stricter security than on many traditional systems. The important differences are:

- A user ID doesn't map to a single login name on a machine; instead it maps to a unique domain name-login name pair.
- A group ID maps to a unique domain name-group name pair.

- Groups and users share the same namespace in the user database. There is no **/etc/passwd** or **/etc/groups** file.
- There is no way for a process to obtain the appropriate permissions to set the effective user or group ID of a process or file.

From a programming standpoint, there are two important consequences. First, functions such as *getpwnam()* and *getgrnam()* which take a login or group name now accept a domain+name pair. Second, another mechanism must be found for *setuid* and *setgid* programs

Domains (*getpwnam()*, *getgrnam()*)

A domain is a collection of Windows NT machines in a network which use a single user database. Most Windows NT workstations therefore have access to at least three domains: one or more domains that have been set up on the network, a local domain which uses the machine's own database, and the built-in domains of every Windows NT machine.

User and group names therefore have the format *domain+name*. The **pw_name** and **gr_name** members in the **passwd** and **group** structures have the format:

domain+name

They can also be passed in the form:

+name
name

These are interpreted as follows:

domain+name

The group or user *name* belonging to the specified *domain*.

name

The group or user *name* in the same domain as the calling process.

+name

The same as *name*: the group or user *name* in the same domain as the calling process.

The passwd Structure

Since the files `/etc/passwd` and `/etc/groups` don't exist, some of the information for the **struct passwd** must be obtained in a non-standard way. The **pw_gecos** (user information) member is filled with the text in the user profile *Description* field, visible through the Win32 User Manager.

The content of the **pw_shell** member is always `/bin/sh`.

Because other applications in both the INTERIX and Win32 environments may use the information in those fields, don't make assumptions about the contents of the **pw_gecos** member.

authenticateuser() and exec*_asuser()

INTERIX provides an interface to execute a process as another user. These functions and structures are defined in the header file `<security.h>`. There is the function `authenticateuser( )` and six `exec*_asuser( )` functions: `execl_asuser( )`, `execle_asuser( )`, `execlp_asuser( )`, `execv_asuser( )`, `execve_asuser( )`, `execvp_asuser( )`.

All of these functions use a user security structure, **struct usersec**, to identify the "other" user. The **usersec** structure looks like this:

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

The meanings of **domain**, **user**, and **password** are self-evident. The **logontype** and **loginprovider** members are provided for future development; leave them at the default values of 0.

To run a process as another user, fill in the members of the structure. Once the structure has been filled in, you can validate the data by passing it to the `authenticateuser( )` function, which returns 0 if the data is valid and 1 if the data is not.

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

Absolute Pathnames

In applications source code, it's common to see pathnames hard-coded to **/bin** or **/usr/local/bin** rather than using *execp()* or *execvp()*; but because INTERIX systems can be installed in any directory on a Windows NT machine, this is a problem. There may not be a **/bin** directory at the root of the device. Though an executable name is the most common absolute pathname, this problem isn't limited to executable files: the directory **/etc** may not be at the root of the device either, or **/usr/spool**.

INTERIX provides a partial solution by providing the standard directories within the directory where the INTERIX package is installed: you know that there is a **/usr**, a **/bin**, a **/tmp** within that directory tree.

There are three solutions, with varying degrees of portability:

- If the executable will run only on your machine, replace the string with the path to the appropriate path for your system. This is essentially non-portable, and not recommended.
- If you're only concerned with making the code run on INTERIX machines, use an `#ifdef __OPENNT` to isolate the pathname definition and rebuild the pathname using the supplied library functions *_getInstallPath()* or *_prefixInstallPath()*; prototypes are in **<unistd.h>**. For most porting applications, this is sufficient. (Because these are nonstandard APIs, you'll also have to `#define _ALL_SOURCE` in your code.) This is discussed below.
- For executables, if you want the code to be portable back to other POSIX-conformant platforms and you don't want to depend upon **#ifdefs**, you can write a function that uses POSIX *confstr()* to find the value of `_CS_PATH`, then iterates through the components of `_CS_PATH`, searching for the executable.

The *_getInstallPath()* function returns a pointer to a static buffer that contains the full pathname where the INTERIX product was installed. The prototype is:

```
char * _getInstallPath( void );
```

The *_prefixInstallPath()* function constructs a full path, rooted at the install path and extended by the path argument. For example, suppose the INTERIX product is installed in **//C/SSI/** and you want to extend the path **/var/spool/mail**. The function prototype is:

```
char * _prefixInstallPath(char *path, char *buf, int len);
```

The call `_prefixInstallPath("/var/spool/mail", NULL, 0)` returns a pointer to the string `"/C/SSI/var/spool/mail"`.

In fact, there is no **/var** directory under the INTERIX install hierarchy, so you might have to change the directory to **/usr/spool/mail**.

If *buf* is non-NULL, the path is stored in *buf* (maximum size *len*). If *buf* is NULL, the path is constructed in a static buffer, and that address is returned. The function returns NULL for errors.

Further Reading

The following references are useful in understanding the POSIX.1 programming environment:

Lewine, Donald. *POSIX Programmer's Guide*. O'Reilly, 1991. ISBN 0-937175-73-0.

Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. ISBN 0-201-56317-7

Zlotnick, Fred. *The POSIX.1 Standard: a programmer's guide*. Benjamin / Cummings, 1991. ISBN 0-8053-9605

INTRODUCTION

AN OVERVIEW OF CURSES

USING THE CURSES LIBRARY

FUNCTION DESCRIPTIONS

COMPATIBILITY WITH OLDER VERSIONS

FURTHER READING

CHAPTER 5

Programming with Curses

The implementation of the curses library shipped with INTERIX is the Ncurses package. This chapter is taken from the documentation for Ncurses, and was originally written by Eric S. Raymond and Zeyd M. Ben-Halim.

It has been shortened and modified to reflect the INTERIX implementation. For the complete text, see the Adobe Acrobat file **`$OPENNT_ROOT/pubs/curses.pdf`**.

Introduction

This is an introduction to programming with curses. It is not an exhaustive reference for the curses Application Programming Interface (API); that role is filled by the curses manual pages. If you are already an experienced curses programmer, you should nevertheless read the sections on Debugging, Compatibility with Older Versions, and Hints, Tips, and Tricks. These will bring you up to speed on the special features and quirks of the ncurses implementation. If you are not so experienced, keep reading.

The curses package is a subroutine library for terminal-independent screen-painting and input-event handling which presents a high level screen model to the programmer, hiding differences between terminal

types and doing automatic optimization of output to change one screenful of text into another.

Curses uses terminfo, which is a database format that can describe the capabilities of thousands of different terminals.

Scope of This Document

This document describes Ncurses, a freeware implementation of the System V curses API with some clearly marked extensions. It includes the following System V curses features:

- Support for multiple screen highlights (BSD curses could only handle one ‘standout’ highlight, usually reverse-video)
- Support for line- and box-drawing using forms characters
- Recognition of function keys on input
- Color support
- Support for pads (windows of larger than screen size on which the screen or a subwindow defines a viewport)

Also, this package makes use of the insert and delete line and character features of terminals so equipped, and determines how to optimally use these features with no help from the programmer. It allows arbitrary combinations of video attributes to be displayed, even on terminals that leave “magic cookies” on the screen to mark changes in attributes.

Terminology

The following terminology is used here with reasonable consistency:

window

A data structure describing a sub-rectangle of the screen (possibly the entire screen). You can write to a window as though it were a miniature screen, scrolling independently of other windows on the physical screen.

screens

A subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower

right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

terminal screen

The package's idea of what the terminal display currently looks like, i.e., what the user sees now. This is a special screen.

An Overview of Curses

Compiling Programs Using Curses

In order to use the library, certain types and variables must be defined. The programmer must have a line:

```
#include <curses.h>
```

at the top of the program source. The screen package uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. **<curses.h>** also includes **<termios.h>**. It is redundant (but harmless) for the programmer to do these includes, too. In linking with curses you need **-lcurses** in your LDFLAGS or on the command line. There is no need for any other libraries.

Updating the Screen

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named **WINDOW** is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) coordinates of the upper left hand corner) and its size. One of these (called *curscr*, for current screen) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for standard screen) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen; it's more like a scratchpad or write buffer.

To make the section of physical screen corresponding to a window reflect the contents of the window structure, the routine *refresh()* (or

wrefresh() if the window is not *stdscr*) is called.

A given physical screen section may be within the scope of any number of overlapping windows. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say “make it look like this,” and let the package implementation determine the most efficient way to repaint the screen.

Standard Windows and Function Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never actually access *curscr* directly. Changes should be made to through the API, and then the routine *refresh()* (or *wrefresh()*) called.

Many functions are defined to use *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character as argument. To write to a different window, use the routine *waddch()* (for ‘w’indow-specific *addch()*) is provided. This convention of prepending function names with a ‘w’ when they are to be applied to specific windows is consistent. The only routines which do not follow it are those for which a window must always be specified.

In order to move the current (y, x) coordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix ‘mv’ and the desired (y, x) coordinates prepended to the arguments to the function. For example, the calls

```
move(y, x);  
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);  
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (*y, x*) coordinates. If a function requires a window pointer, it is always the first parameter passed.

Variables

The **curses** library sets some variables describing the terminal capabilities.

type	name	description
int	<i>LINES</i>	number of lines on the terminal
int	<i>COLS</i>	number of columns on the terminal

The **<curses.h>** also introduces some **#define** constants and types of general usefulness:

bool	boolean type, actually a char (e.g., bool doneit);
TRUE	boolean 'true' flag (1)
FALSE	boolean 'false' flag (0)
ERR	error flag returned by routines on a fail (-1)
OK	error flag returned by routines when things go right

Using the Curses Library

Now we describe how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. These instructions will work on any window, providing you change the function names and parameters as mentioned above.

Here is a sample program to motivate the discussion:

```
#include <curses.h>
#include <signal.h>
static void finish(int sig);
main(int argc, char *argv[])
{
    /* initialize non-curses data structures here */
    /* arrange interrupts to terminate */
    (void) signal(SIGINT, finish);
    /* initialize the curses library */
    (void) initscr();
    /* enable keyboard mapping */
    keypad(stdscr, TRUE);
    /* tell curses not to do NL->CR/NL on output
```

```

        */
(void) nonl();
    /* take input chars one at a time, no wait for
       \n */
(void) cbreak();
    /* don't echo input */
(void) noecho();
if (has_colors())
{
    start_color();
    /*
     * Simple color assignment,
     * often all we need.
     */
    init_pair(COLOR_BLACK, COLOR_BLACK,
              COLOR_BLACK);
    init_pair(COLOR_GREEN, COLOR_GREEN,
              COLOR_BLACK);
    init_pair(COLOR_RED, COLOR_RED, COLOR_BLACK);
    init_pair(COLOR_CYAN, COLOR_CYAN,
              COLOR_BLACK);
    init_pair(COLOR_WHITE, COLOR_WHITE,
              COLOR_BLACK);
    init_pair(COLOR_MAGENTA, COLOR_MAGENTA,
              COLOR_BLACK);
    init_pair(COLOR_BLUE, COLOR_BLUE,
              COLOR_BLACK);
    init_pair(COLOR_YELLOW, COLOR_YELLOW,
              COLOR_BLACK);
}
for (;;)
{
    /* refresh, accept single keystroke of input */
    int c = getch();
    /* process the command keystroke */
}
finish(0);                /* we're done */
}
static void finish(int sig)
{
    endwin();
    /* do your non-curses wrapup here */
    exit(0);
}

```

Starting Up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. The function *initscr()* does both these things. Since it must allocate space for the windows, it can overflow memory when attempting to do so. On the rare occasions this happens, *initscr()* will terminate the program with an error message. *initscr()* must always be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *cbreak()* should be called after *initscr()*.

Once the screen windows have been allocated, you can set them up for your program. If you want to, say, allow a screen to scroll, use *scrollok()*. If you want the cursor to be left in place after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) coordinates after updating it.

You can create new windows of your own using the functions *newwin()*, *derwin()*, and *subwin()*. The routine *delwin()* will allow you to get rid of old windows. All the options described above can be applied to any window.

Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*.

addch() adds a character at the current (y, x) coordinates. *move()* changes the current (y, x) coordinates to whatever you want them to be. It returns ERR if you try to move off the window. As mentioned above, you can combine the two into *mvaddch()* to do both things at once.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with

an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr* as its argument, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

Input

The complementary function to *addch()* is *getch()* which, if *echo()* is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. Since initially the terminal has echoing enabled and is in ordinary “cooked” mode, one or the other has to be changed before calling *getch()*; otherwise, the program’s output will be unpredictable.

When you need to accept line-oriented input in a window, the functions *wgetstr()* and friends are available. There is even a *wscanw()* function that can do *scanf()*-style multi-field parsing on window input. These pseudo-line-oriented functions turn on echoing while they execute.

The example code above uses the call *keypad(stdscr, TRUE)* to enable support for function-key mapping. With this feature, the *getch()* code watches the input stream for character sequences that correspond to arrow and function keys. These sequences are returned as pseudo-character values. The **#define** values returned are listed in the **< curses.h >**. The mapping from sequences to **#define** values is determined by **key_** capabilities in the terminal’s terminfo entry.

Using Forms Characters

The *addch()* function (and some others, including *box()* and *border()*) can accept some pseudo-character arguments which are specially defined by **ncurses**. These are **#define** values set up in the **< curses.h >** header; see there for a complete list (look for the prefix **ACS_**).

The most useful of the ACS defines are the forms-drawing characters. You can use these to draw boxes and simple graphs on the screen. If the terminal does not have such characters, **< curses.h >** will map them to a recognizable (though ugly) set of ASCII defaults.

Character Attributes and Color

The ncurses package supports screen highlights including standout, reverse-video, underline, and blink. It also supports color, which is treated as another kind of highlight.

Highlights are encoded, internally, as high bits of the pseudo-character type (**chtype**) that `< curses.h >` uses to represent the contents of a screen cell. See the `< curses.h >` header file for a complete list of highlight mask values (look for the prefix `A_`).

There are two ways to make highlights. One is to logical-or the value of the highlights you want into the character argument of an `addch()` call, or any other output call that takes a `chtype` argument.

The other is to set the current-highlight value. This is logical-or'ed with any highlight you specify the first way. You do this with the functions `attron()`, `attroff()`, and `attrset()`; see the manual pages for details.

Color is a special kind of highlight. The package actually thinks in terms of color pairs, combinations of foreground and background colors. The sample code above sets up eight color pairs, all of the guaranteed-available colors on black. Note that each color pair is, in effect, given the name of its foreground color. Any other range of eight nonconflicting values could have been used as the first arguments of the `init_pair()` values.

Once you've done an `init_pair()` that creates color-pair *N*, you can use `COLOR_PAIR(N)` as a highlight that invokes that particular color combination. Note that `COLOR_PAIR(N)`, for constant *N*, is itself a compile-time constant and can be used in initializers.

Finishing Up

In order to clean up after the ncurses routines, the routine `endwin()` is provided. It restores tty modes to what they were when `initscr()` was first called, and moves the cursor down to the lower-left corner. Thus, anytime after the call to `initscr`, `endwin()` should be called before exiting.

Function Descriptions

We describe the detailed behavior of some important curses functions here, as a supplement to the reference page descriptions.

Initialization and Wrapup

initscr()

The first function called should almost always be *initscr()*. This will determine the terminal type and initialize curses data structures. *initscr()* also arranges that the first call to *refresh()* will clear the screen. If an error occurs a message is written to standard error and the program exits. Otherwise it returns a pointer to *stdscr*. A few functions may be called before *initscr()* (*slk_init()*, *filter()*, *ripofflines()*, *use_env()*, and, if you are using multiple terminals, *newterm()*.)

endwin()

Your program should always call *endwin()* before exiting or shelling out of the program. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper nonvisual mode. Calling *refresh()* or *doupdate()* after a temporary escape from the program will restore the ncurses screen from before the escape.

newterm(type, ofp, ifp)

A program which outputs to more than one terminal should use *newterm()* instead of *initscr()*. *newterm()* should be called once for each terminal. It returns a variable of type **SCREEN *** which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and **FILE** pointers for the output and input of the terminal. If type is NULL then the environment variable *TERM* is used. *endwin()* should be called once at wrapup time for each terminal opened using this function.

set_term(new)

This function is used to switch to a different terminal previously opened by *newterm()*. The screen reference for the new terminal is passed as the parameter. The previous terminal is returned by the function. All other calls affect only the current terminal.

delscreen(sp)

The inverse of *newterm()*; deallocates the data structures associated with a given **SCREEN** reference.

Causing Output to the Terminal

refresh()* and *wrefresh(win)

These functions must be called to actually get any output on the terminal, as other routines merely manipulate data structures. *wrefresh()* copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. *refresh()* does a refresh of *stdscr*. Unless *leaveok()* has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

doupdate()* and *wnoutrefresh(win)

These two functions allow multiple updates with more efficiency than *wrefresh()*. To use them, it is important to understand how curses works. In addition to all the window structures, curses keeps two data structures representing the terminal screen: a physical screen, describing what is actually on the screen, and a virtual screen, describing what the programmer wants to have on the screen. *wrefresh()* works by first copying the named window to the virtual screen (*wnoutrefresh()*), and then calling the routine to update the screen (*doupdate()*). If the programmer wishes to output several windows at once, a series of calls to *wrefresh()* will result in alternating calls to *wnoutrefresh()* and *doupdate()*, causing several bursts of output to the screen. By calling *wnoutrefresh()* for each window, it is then possible to call *doupdate()* once, resulting in only one burst of output, with fewer total characters transmitted (this also avoids a visually annoying flicker at each update).

Low-Level Capability Access

setupterm(term, filenum, errret)

This routine is called to initialize a terminal's description, without setting up the curses screen structures or changing the tty-driver mode bits. *term* is the character string representing the name of the terminal being used. *filenum* is the file descriptor of the terminal to be used for output. *errret* is a pointer to an integer, in which a success or failure indication is returned. The values returned can

be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the terminfo database).

The value of *term* can be given as NULL, which will cause the value of *TERM* in the environment to be used. The *errret* pointer can also be given as NULL, meaning no error code is wanted. If *errret* is defaulted, and something goes wrong, *setupterm*() will print an appropriate error message and exit, rather than returning. Thus, a simple program can call *setupterm*(0, 1, 0) and not worry about initialization errors.

After the call to *setupterm*(), the global variable *cur_term* is set to point to the current structure of terminal capabilities. By calling *setupterm*() for each terminal, and saving and restoring *cur_term*, it is possible for a program to use two or more terminals at once. *Setupterm*() also stores the names section of the terminal description in the global character array *ttytype*[]. Subsequent calls to *setupterm*() will overwrite this array, so you'll have to save it yourself if need be.

Debugging

These functions are not part of the standard curses API!

trace()

This function can be used to explicitly set a trace level. If the trace level is nonzero, execution of your program will generate a file called **trace** in the current working directory containing a report on the library's actions. Higher trace levels enable more detailed (and verbose) reporting—see comments attached to *TRACE_* defines in the **<curses.h>** file for details. (It is also possible to set a trace level by assigning a trace level value to the environment variable *NCURSES_TRACE*).

Hints, Tips, and Tricks

The Ncurses manual pages are a complete reference for this library. In the remainder of this document, we discuss various useful methods that may not be obvious from the manual page descriptions.

Some Notes of Caution

If you find yourself thinking you need to use *noraw()* or *nocbreak()*, think again and move carefully. It's probably better design to use *getstr()* or one of its relatives to simulate cooked mode. The *noraw()* and *nocbreak()* functions try to restore cooked mode, but they may end up clobbering some control bits set before you started your application. Also, they have always been poorly documented, and are likely to hurt your application's usability with other curses libraries.

Bear in mind that *refresh()* is a synonym for *wrefresh(stdscr)*, and don't try to mix use of *stdscr* with use of windows declared by *newwin()*; a *refresh()* call will blow them off the screen. The right way to handle this is to use *subwin()*, or not touch *stdscr* at all and tile your screen with declared windows which you then *wnoutrefresh()* somewhere in your program event loop, with a single *doupdate()* call to trigger actual repainting.

You are much less likely to run into problems if you design your screen layouts to use tiled rather than overlapping windows. Historically, curses support for overlapping windows has been weak, fragile, and poorly documented. The ncurses library is not yet an exception to this rule.

Try to avoid using the global variables *LINES* and *COLS*. Use *getmaxyx()* on the *stdscr* context instead. Reason: your code may be ported to run in an environment with window resizes, in which case several screens could be open with different sizes.

Temporarily Leaving Ncurses Mode

Sometimes you will want to write a program that spends most of its time in screen mode, but occasionally returns to ordinary canonical (cooked) mode. A common reason for this is to support shell-out. This behavior is simple to arrange in ncurses.

To leave ncurses mode, call *endwin()* as you would if you were intending to terminate the program. This will take the screen back to cooked

mode; you can do your shell-out. When you want to return to ncurses mode, simply call *refresh()* or *doupdate()*. This will repaint the screen.

There is a boolean function, *isendwin()*, which code can use to test whether ncurses screen mode is active. It returns TRUE in the interval between an *endwin()* call and the following *refresh()*, FALSE otherwise.

Here is some sample code for shellout:

```
addstr("Shelling out...");
def_prog_mode();      /* save current tty modes */
endwin();             /* restore original tty modes */
system("sh");          /* run shell */
                        /* prepare return message */
addstr("returned.\n");
                        /* restore save modes, repaint screen */
refresh();
```

Handling Multiple Terminal Screens

The *initscr()* function actually calls a function named *newterm()* to do most of its work. If you are writing a program that opens multiple terminals, use *newterm()* directly.

For each call, you will have to specify a terminal type and a pair of file pointers; each call will return a screen reference, and *stdscr* will be set to the last one allocated. You will switch between screens with the *set_term()* call. Note that you will also have to call *def_shell_mode()* and *def_prog_mode()* on each tty yourself.

Testing for Terminal Capabilities

Sometimes you may want to write programs that test for the presence of various capabilities before deciding whether to go into ncurses mode. An easy way to do this is to call *setupterm()*, then use the functions *tigetflag()*, *tigetnum()*, and *tigetstr()* to do your testing.

A particularly useful case of this often comes up when you want to test whether a given terminal type should be treated as 'smart' (cursor-addressable) or 'stupid'. The right way to test this is to see if the return value of *tigetstr("cup")* is non-NULL. Alternatively, you can include the **<term.h>** file and test the value of the macro *cursor_address*.

Tuning for Speed

Use the *addchstr()* family of functions for fast screen-painting of text when you know the text doesn't contain any control characters. Try to make attribute changes infrequent on your screens. Don't use the *immedok()* option!

Special Features of Ncurses

When running on PC-clones, ncurses has enhanced support for the IBM high-half and ROM characters. The *A_ALTCHARSET* highlight, enables display of both high-half ACS graphics and the PC ROM graphics 0-31 that are normally interpreted as control characters.

The *wresize()* function allows you to resize a window in place.

Compatibility with Older Versions

Despite our best efforts, there are some differences between ncurses and the (undocumented!) behavior of older curses implementations. These arise from ambiguities or omissions in the documentation of the API.

The mouse interfacing features found in the ncurses source is not enabled for INTERIX.

Refresh of Overlapping Windows

If you define two windows A and B that overlap, and then alternately scribble on and refresh them, the changes made to the overlapping region under historic curses versions were often not documented precisely.

To understand why this is a problem, remember that screen updates are calculated between two representations of the *entire* display. The documentation says that when you refresh a window, it is first copied to the virtual screen, and then changes are calculated to update the physical screen (and applied to the terminal). But "copied to" is not very specific, and subtle differences in how copying works can produce different behaviors in the case where two overlapping windows are each being refreshed at unpredictable intervals.

What happens to the overlapping region depends on what *wnoutrefresh()* does with its argument — what portions of the argument window it copies to the virtual screen. Some implementations do “change copy,” copying down only locations in the window that have changed (or been marked changed with *wtouchln()* and friends). Some implementations do “entire copy,” copying *all* window locations to the virtual screen whether or not they have changed.

The ncurses library itself has not always been consistent on this score. Due to a bug, versions 1.8.7 to 1.9.8a did entire copy. Versions 1.8.6 and older, and versions 1.9.9 and newer, do change copy.

For most commercial curses implementations, it is not documented and not known for sure (at least not to the ncurses maintainers) whether they do change copy or entire copy. We know that System V release 3 curses has logic in it that looks like an attempt to do change copy, but the surrounding logic and data representations are sufficiently complex, and our knowledge sufficiently indirect, that it’s hard to know whether this is reliable.

It is not clear what the SVR4 documentation and XSI standard intend. The XSI Curses standard barely mentions *wnoutrefresh()*; the SVR4 documents seem to be describing entire-copy, but it is possible with some effort and straining to read them the other way.

It might therefore be unwise to rely on either behavior in programs that might have to be linked with other curses implementations. Instead, you can do an explicit *touchwin()* before the *wnoutrefresh()* call to guarantee an entire-contents copy anywhere.

The really clean way to handle this is to use the panels library. If, when you want a screen update, you do *update_panels()*, it will do all the necessary *wnoutrefresh()* calls for whatever panel stacking order you have defined. Then you can do one *doupdate()* and there will be a *single* burst of physical I/O that will do all your updates.

XSI Curses Conformance

The ncurses library is intended to be base-level conformant with the XSI Curses standard from X/Open. May extended-level features (in fact, almost all features not directly concerned with wide characters and internationalization) are also supported.

Erased areas of a window are filled with a blank with the normal attribute, unless and until modified by the functions *bkgdset()* or *wbkgdset()*.

Also, ncurses meets the XSI requirement that every macro entry point have a corresponding function which may be linked (and will be prototype-checked) if the macro definition is disabled with **#undef**.

Further Reading

For information on the curses library calls, as well as those of the termcap and terminfo libraries, see the reference pages.

For more information on programming with the curses library, see the book *Programming with curses*, by John Strang, published by O'Reilly & Associates, Inc.

THE SERVICE INTERFACE

SERVICE EXECUTION ENVIRONMENT

CODING SERVICE APPLICATIONS

MESSAGE LOGGING

CHAPTER 6

Services

This chapter discusses writing Windows NT services for the INTERIX subsystem.

On traditional systems, a daemon is a process without a controlling terminal that lives for a long time. A Windows NT service is a background process that is not terminated by logging out; it's the Windows NT equivalent to a daemon process. In order to port daemons to the INTERIX platform, the INTERIX subsystem provides the service interface.

The Service Interface

The program *psxrun.exe* is a Win32 application that mediates for POSIX processes that need access to the Windows NT service control manager. INTERIX services can be administered using the **Services** applet of the Control Panel 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 username and password must be provided when the service is installed. If no username is provided, it defaults to the user "LocalSystem" which is an administrative login without network access.

When a request comes from either the Services applet or *service* to kill a particular service, *psxrun.exe* sends the signal SIGTERM to the service.

(Some daemons use signals for interprocess communication; SIGTERM is unlikely to interfere with those daemons.)

Service Execution Environment

Services on Windows NT run in a minimal environment. When a service is started, it inherits the environment of the Service Control Manager (normally the system environment) and defines *HOME*, *LOGNAME*, and *TZ*.

Here's how *psxrun.exe* assigns values for those environment variables:

- *LOGNAME* is based on *USERNAME* as set up by Windows NT; this is given as an argument to the *service* program when the service is installed; if no username is given, it defaults to the user LocalSystem.
- *HOME* is built from *HOMEPATH* and *HOMEDRIVE*, provided by the operating system for the specified user.
- *TZ* is your timezone, derived from the operating system. The format is documented in the *tzset()* reference page.
- The *PATH* inherited from the Service Control Manager is copied into the environment variable *_PSXLIBPATH*; the contents of *PATH* are then converted into POSIX format.

Some system parameters such as *SYSTEMROOT* and *NUMBER_OF_PROCESSORS* are also available.

A lot of daemons run as a particular user using *setuid()* or *seteuid()*. Because of the security on Windows NT, this won't work. You may have to **#ifdef** out that code and replace it with a simple *exec_asuser()* call, or install the service as a particular user.

Coding Service Applications

There are some differences between writing traditional daemons and Windows NT services. Some of the rules for coding daemons will cause problems with the service interface, specifically:

1. *Fork()* and have the parent *exit()*

Don't. The intent is to make the shell think the command is done and guarantee that the child is not a process group leader. If the parent process *exit()*s, *psxrun.exe* will treat the program as having exited, and Windows NT's Service Control Manager will report that the service was never successfully started.

2. Call *setsid()* to create a new session

Don't. The call will fail. The *psxrun.exe* program ensures that the service process is a session leader.

3. Use hard-coded absolute pathnames such as */tmp*, */bin/sh*, or */usr/spool*.

Don't. Instead, use *_getInstallPath()* or *_prefixInstallPath()* to construct pathnames to system files in */etc*, */usr*, */bin*, and so on.

4. Access network drives.

Don't. Network drives are typically mounted on drive letters when a user logs in to the console, 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* use 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 your services must access a network drive, then you should reserve certain drive letters for exclusive use by the system, and ensure that interactive users know that they shouldn't use these drive letters.

5. Your service should exit when it receives the SIGTERM signal. If necessary, you can catch the SIGTERM signal, clean up, and shut down. It is bad form to spend too long cleaning up; there may be interactions with the Service Control Manager. (In this case, "too long" means more than a few seconds.)

The library contains the *daemon()* application (also found on BSD systems), which makes creating service applications easier.

THE DIRECTORY HIERARCHY

TOOLS

LIBRARIES AND HEADER FILES

BUILDING AN X WINDOW APPLICATION

THE INTERIX MOTIF DEVELOPMENT KIT

CHAPTER 7

Building X Window Applications

This chapter describes the basics of building X Window Applications using INTERIX. The INTERIX Software Development Kit contains the X11R5 libraries and header files and some tools specifically for building X Window utilities.

Several of the tools provided as part of the Software Development Kit (*cpp*, *imake*, *mkmf*, *mkdirhier*, *makedepend*) are necessary to build X11 software packages provided by third parties. The *cpp* implementation in particular is intended only for use by *imake*, not by the C compiler.

The X Directory Hierarchy

The install program sets up the following directory hierarchy under **\$OPENNT_ROOT/usr/X11R5**:

- bin**
- include/**
- include/X11/**
- lib/X11/app-defaults/**
- lib/X11/config/**
- lib/X11/nls/**
- lib/X11/twm/**
- lib/X11/x11perfcomp/**

Most code for X Window utilities assumes a directory named **/usr/X11** rather than **/usr/X11R5**. To deal with this, you can either:

- Change the application code.
- Create a shadow directory named **X11**. The shadow directory is populated with either links to or copies of the header files and libraries. Links are possible if the INTERIX Software Development Kit is installed on an NTFS file system. If it is installed on NTFS, you may create links.

Utilities

Most third-party X11 source packages are built using the tools *imake* and *xmkmf* to process configuration files that, in turn, generate **Makefiles**. Both *imake* and *xmkmf* are included in the INTERIX Software Development Kit.

The *xmkmf* utility calls *imake* to process the **Imakefile**; it produces a **Makefile**. The configuration information used by *xmkmf* (and by *imake*) is installed in the directory **\$OPENNT_ROOT/usr/x11/lib/config**.

If you have an unusual INTERIX installation, you may need to edit the file **OpenNT.cf** in that directory. In particular, ensure that the entries for ProjectRoot, BourneShell, and CppCmd reflect where INTERIX was installed. If your INTERIX root is **//D/progs/interix**:

```
#define ProjectRoot    //D/progs/interix/usr/X11R5
#define BourneShell    //D/progs/interix/bin/ksh
#define CppCmd         //D/progs/interix/usr/contrib/cpp
```

If you are using *cc* or *c89*, you must also change the value of the CcCmd macro from *gcc* to *cc* or *c89*.

Libraries and Header Files

The X11 libraries provided in the INTERIX Software Development Kit are static libraries. This means that the resultant executables are large.

Building an Application

The X11 header directory must be part of the object compilation:

```
CFLAGS = -O -I/OpenNT/usr/X11R5/include
```

The libraries must be included and be after the object files:

```
XLIBS = -lXaw -lXmu -lXt -lXext -lX11
```

General purpose libraries have to be included *after* the X11 libraries:

```
XLIBS = -lXaw -lXmu -lXt -lXext -lX11
```

The `-L` directive (library location) must *precede* the object file

```
XLIBDIR = -L/OpenNT/usr/X11R5/lib
```

Some applications (such as the example *xtetris* source) use the *xrdb* utility in order to create the application defaults files; to build these applications, you must set your display and start the X11 Server before you start the build process.

Many X Window applications come with **Imakefiles**. You can create a **Makefile** from an **Imakefile** using either *xmkmf* or *imake*. (If you are building a Motif application using the INTERIX Motif Development Kit, use *mxmkmf*.)

Using *xmkmf* to build an application from an **Imakefile**, you'll see output like this:

```
$ xmkmf
mv Makefile Makefile.bak
imake -DUseInstalled -I/OpenNT/usr/X11R5/lib/X11/
config
```

You can use *imake* directly. You may have to pass appropriate **-D** and **-I** options. The same command as an *imake* command:

```
$ imake -DUseInstalled -I/OpenNT/usr/X11R5/lib/X11/
config
```

Once the **Makefile** has been built, you can build the application with *make*.

Note: Some X Window applications use *xrdb* as part of the build process (in the recipe for **make install**, for example). For *xrdb* to run properly, you must have the X11 Server running. This is true of the *xtetris* example source.

The INTERIX Motif Development Kit

When you install the INTERIX Motif Development Kit, you are installing the MWM window manager, some utilities, and the Motif 1.2.4 libraries and header files.

The INTERIX MDK provides its own version of *xmkmf*, called *mxmkmf*, which is used in the same way.

CHAPTER 8

System Error Codes

This section provides an overview of the system error returns. Because the INTERIX subsystem does not necessarily use the same error numbers as traditional systems, you should always use the symbolic names defined in **<errno.h>**.

Nearly all of the system calls can return an error number in the external variable *errno*, which is also defined in **<errno.h>**

When a system call detects an error, it returns an integer value indicating failure (usually -1) and sets the variable *errno* accordingly. Successful calls never set *errno*; once set, it remains until another error occurs. It should only be examined after an error.

When a system call returns -1, the calling function can interpret the failure and take action accordingly. A number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

The following is a complete list of the errors and their names as given in **<errno.h>**. They are listed in alphabetical order by symbolic name.

According to POSIX, a system can return error codes other than the ones listed here; a conforming application should be prepared to deal with other “unknown” error codes.

[E2BIG] *Arg list too long.*

The number of bytes used for the argument and environment list of the new process exceeded the current limit of {ARG_MAX} in `<limits.h>`.

[EACCES] *Permission denied.*

An attempt was made to access a file in a way forbidden by its file access permissions.

[EADDRINUSE] *Address already in use.*

Normally, each address can only be used once.

[EADDRNOTAVAIL] *Address cannot be assigned.*

Normally results from an attempt to create a socket with an address not on this machine.

[EAFNOSUPPORT] *Address family not supported by protocol family.*

The address was incompatible with the requested protocol.

[EAGAIN] | [EWOULDBLOCK] *Resource temporarily unavailable.*

This is a temporary condition and later calls to the same routine may complete normally.

[EALREADY] *Operation already in progress.*

An operation was attempted on a non-blocking object that already had an operation in progress.

[EBADF] *Bad file descriptor.*

A file descriptor argument was out of range, referred to no open file, or a read (write) request was made to a file that was only open for writing (reading).

[EBUSY] *Resource busy.*

An attempt to use a system resource which was in use at the time in a manner which would have conflicted with the request.

[ECHILD] *No child processes.*

A `wait()` or `waitpid()` function was executed by a process that had no existing or unwaited-for child processes.

[ECONNABORTED] *Software caused connection abort.*

A connection abort was caused internal to your host machine

[ECONNREFUSED] *Connection refused.*

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

[ECONNRESET] *Connection reset by peer.*

A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

[EDEADLK] | [EDEADLOCK] *Resource deadlock avoided.*

An attempt was made to lock a system resource that would have resulted in a deadlock situation.

[EDESTADDRREQ] *Destination address required.*

An operation on a socket requires a destination address.

[EDOM] *Numerical argument out of domain.*

A numerical input argument was outside the defined domain of the mathematical function.

[EDQUOT] *Disk quota exceeded.*

One of the following failed because the user's quota of disk space was exhausted: a write to an ordinary file, the creation of a directory or the creation of a directory entry.

[EEXIST] *File exists.*

An existing file was mentioned in an inappropriate context, for instance, as the new link name in a *link()* function.

[EFAULT] *Bad address.*

The system detected an invalid address in attempting to use an argument of a call.

[EFBIG] *File too large.*

The size of a file exceeded the maximum size, defined for Windows NT as $2^{32}-1$ bytes.

[EFTYPE] *Inappropriate file type or format.*

An operation was attempted on a file whose type or format would not support it..

[EHOSTDOWN] *Host is down.*

A socket operation failed because the destination host was down.

[EHOSTNOTFOUND] *Host could not be found.*

A socket operation was attempted to a host that could not be located.

[EHOSTUNREACH] *No route to host.*

A socket operation was attempted to an unreachable host.

[EILSEQ] *Invalid multi-byte character.*

The call contained an multi-byte character that was invalid in the current locale.

[EINPROGRESS] *Operation now in progress.*

An operation that takes a long time to complete, such as a *connect()*, was attempted on a non-blocking object.

[EINTR] *Interrupted function call.*

An asynchronous signal (such as SIGINT) was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call will seem to have returned the error condition.

[EINVAL] *Invalid argument.*

Some invalid argument was supplied. (For example, specifying an undefined signal to a *signal()* or *kill()* function).

[EIO] *Input/output error.*

Some physical input or output error occurred. This error will not be reported until a subsequent operation on the same file descriptor and may be lost (over written) by any subsequent errors.

[EISCONN] *Socket is already connected.*

A connect request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination when already connected.

[EISDIR] *Is a directory.*

An attempt was made to open a directory with write mode specified.

[EMFILE] *Too many open files.*

The number of files open exceeds {OPEN_MAX}.

[EMLINK] *Too many links.*

Maximum allowable hard links to a single file has been exceeded (limit depends upon the filesystem; a POSIX-conformant filesystem-

tem supports at least 8). You can check the maximum number of links allowed on the system with the command *getconf LINK_MAX*.

[EMSGSIZE] *Message too long.*

Message sent on a socket was too long for an internal buffer or another limit.

[ENAMETOOLONG] *File name too long.*

File name exceeded {PATH_MAX} or a component of a file name exceeded {NAME_MAX}. For truly portable software, it may not be safe to rely on hard-coded values of {NAME_MAX} and {PATH_MAX}; on a networked system, the values may change depending on the filesystem. A safer way to check for the values on the current filesystem is to use *pathconf()* or *fpathconf()* before constructing a file name.

[ENETDOWN] *Network is down.*

A socket operation encountered a dead network.

[ENETRESET] *Network dropped connection on reset.*

The host you were connected to crashed and rebooted.

[ENETUNREACH] *Network is unreachable.*

A socket operation was attempted to an unreachable network.

[ENFILE] *Too many open files in system.*

Maximum number of file descriptors allowable on the system has been reached and a requests for an open cannot be satisfied until at least one has been closed.

[ENOBUFS] *No buffer space available.*

An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

[ENODATA] *No message available.*

No data message was available.

[ENODEV] *Operation not supported by device.*

An attempt was made to apply an inappropriate function to a device, for example, trying to read a write-only device such as a printer.

[ENOENT] *No such file or directory.*

A component of a specified pathname did not exist, or the path-name was an empty string.

[ENOEXEC] *Exec format error.*

A request was made to execute a file that, although it has the appropriate permissions, was not in the format required for an executable file.

[ENOLCK] *No locks available.*

No locks were available.

[ENOMEM] *Cannot allocate memory.*

The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints.

[ENOPROTOPT] *Protocol not available.*

A bad option or level was specified for `getsockopt( )` or `setsockopt( )`.

[ENOSPC] *Device out of space.*

A `write( )` to an ordinary file, the creation of a directory, or the creation of a directory entry failed because no more disk blocks were available on the file system, or the allocation of a file serial number for a newly created file failed because no more numbers were available on the file system.

[ENOSYS] *Function not implemented.*

Attempted a system call that was not available on this system.

[ENOTBLK] *Not a block device.*

A block device operation was attempted on a non-block device or file.

[ENOTCONN] *Socket is not connected.*

An request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.

[ENOTDIR] *Not a directory.*

A component of the specified pathname existed, but it was not a directory, when a directory was expected.

[ENOTEMPTY] *Directory not empty.*

An operation such as an unlink was attempted on a directory that was not empty.

[ENOTSOCK] *Not a socket.*

A socket function was attempted on a descriptor that was not a socket.

[ENOTTY] *Inappropriate I/O control operation for device.*

A control function was attempted for a file or special device for which the operation was inappropriate.

[ENXIO] *No such device or address.*

Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape is not on-line or no disk is loaded in a drive.

[EOPNOTSUPP] *Operation not supported on socket.*

The type of socket (either address family or protocol) does not support the operation.

[EPERM] *Operation not permitted.*

An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resources.

[EPFNOSUPPORT] *Protocol family not supported.*

The protocol family has not been configured or no implementation for it exists.

[EPIPE] *Broken pipe.*

A write on a socket, pipe or FIFO for which there is no process to read the data.

[EPROCLIM] *Too many processes.*

The operation would cause the system to exceed the limit on the number of processes.

[EPROTONOSUPPORT] *Protocol not supported.*

The system does not support the specified protocol.

[EPROTOTYPE] *Protocol wrong for socket.*

The requested protocol doesn't support the semantics of the socket.

[ERANGE] *Numerical result out of range.*

A numerical result of the function was too large to fit in the available space (perhaps exceeded precision).

[EROFS] *Read-only file system.*

An attempt was made to modify a file or directory was made on a file system that was read-only at the time.

[ESHUTDOWN] *Cannot send after socket shutdown.*

A request to send data was disallowed because the socket had already been shut down.

[ESOCKTNOSUPPORT] *Socket type not supported.*

The system does not support the specified socket type.

[ESPIPE] *Illegal seek.*

An *lseek()* function was issued on a socket, pipe or FIFO.

[ESRCH] *No such process.*

No process could be found corresponding to that specified by the given process ID.

[ETIMEDOUT] *Operation timed out.*

A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

[ETOOMANYREFS] *Too many references.*

There were too many references; the system could not splice (System V).

[ETXTBUSY] *Unkown error.*

On INTERIX, this is an unknown error. Traditionally, it means that an attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt was made to open for writing a pure-procedure program while it was being executed.

[EXDEV] *Improper link.*

A hard link to a file on another file system was attempted.

Index

Symbols

`$(CFLAGS)` macro 22
`$OPENNT_ROOT/bin` directory 18
`$OPENNT_ROOT/usr/contrib/bin` directory 18
`$OPENNT_ROOT/usr/share/terminfo` 37
`$OPENNT_ROOT/usr/x11/lib/config/`
 OpenNT.cf file 66
`$OPENNT_ROOT/usr/X11R5/bin` directory 18
`$OPENNT_ROOT/usr/contrib` directory 18
`$OPENNT_ROOT/pubs/GLPL.TXT` 16
`$OPENNT_ROOT/usr/contrib/alpha-pc-`
 opennt/cygnus-2. 16
`$OPENNT_ROOT/usr/contrib/i386-pc-opennt/`
 cygnus-2.7 16
`/bin` directory 41. *See also* `/OpenNT/bin`
 directory
`/Interix/usr/include` directory 20
`/Interix/usr/lib` directory 20
`/usr/local/bin` directory 41
`<errno.h>` 69
`<limits.h>` 36, 70
`<security.h>` 40
`<stdarg.h>` 34–35
`<strings.h>` 36
`<sys/param.h>` 27
`<termios.h>` 29
`<varargs.h>` 34–35. *See also* `<stdarg.h>`
[EADDRINUSE] 70
[EADDRNOTAVAIL] 70
[EAFNOSUPPORT] 70
[ECONNABORTED] 70
[ECONNREFUSED] 71
[ECONNRESET] 71
[EDESTADDRREQ] 71
[EHOSTDOWN] 71
[EHOSTUNREACH] 72
[EISCONN] 72
[EMSGSIZE] 73
[ENETDOWN] 73
[ENETRESET] 73
[ENETUNREACH] 73
[ENOBUFS] 73
[ENOPROTOOPT] 74

[ENOTCONN] 74
[ESHUTDOWN] 76
[ETIMEDOUT] 76
__STDC__ 21
_CS_PATH 27
_getInstallPath() 63
_prefixInstallPath() 63
_PSXLIBPATH environment variable 62

A

abort() 32
access control entry. *See* ACE
access control list. *See* ACL
addch() 46, 49
addchstr() 57
addstr() 49
alarm() 32
_ALL_SOURCE 36, 41
API namespace 36
ar utility 13
ARG_MAX 26, 70
as utility 13
attroff() 51
attron() 51
attrset() 51
authenticateuser() 40

B

bcmp() 37
bcopy() 37
bkgdset() 58
border() 50
box() 50
bzero() 37

C

c89 utility 8, 13, 21, 22, 66
 options 19–20
C89_COMPILER environment variable 20
C89_ECHO environment variable 21

C89_LINKER environment variables 20
 cbreak() 49
 cc utility 8, 13, 19, 21, 66
 cfgetispeed() 29
 cfgetospeed() 29
 cfsetispeed() 29
 cfsetospeed() 29
 CHILD_MAX 26
 chtype 51
 ci utility 13
 co utility 13
 COLOR_PAIR() 51
 compiling 19
 confstr() 27, 41
 controlling terminal, acquiring 38
 cooked mode 55
 cpp utility 14
 _CS_PATH 41
 curscr 45
 curses 43–59
 \$OPENNT_ROOT/pubs/curses.pdf 43
 blink 51
 color 51
 forms-drawing characters 50
 mouse support 57
 multiple terminal screens 56
 reverse-video 51
 sample program 47
 screen redraws 50
 standout 51
 supporting shell-out 55
 underline 51
 variables 47
 XSI curses conformance 58
 <curses.h> 45, 47

D

daemon() 63
 data types
 WINDOW 45
 def_prog_mode() 56
 def_shell_mode() 56
 defined symbols
 __STDC__ 20
 POSIX 21

 __OPENNT 21
 delscreen() 53
 derwin() 49
 directories
 \$OPENNT_ROOT/bin 18
 \$OPENNT_ROOT/usr/contrib/bin 18
 \$OPENNT_ROOT/usr/share/terminfo 37
 \$OPENNT_ROOT/usr/X11R5/bin 18
 \$OPENNT_ROOT/usr/contrib 18
 \$OPENNT_ROOT/usr/contrib/alpha-pc-opennt/cygnus-2. 16
 \$OPENNT_ROOT/usr/contrib/i386-pc-opennt/cygnus-2.7 16
 /Interix/usr/include 20
 /Interix/usr/lib 20
 DISABLE_GETS_WARNING environment variable 28
 domains 39
 domains and login names 38–40
 doupdate() 52, 53

E

[E2BIG] 70
 [EACCES] 70
 [EAGAIN] 70
 [EALREADY] 70
 [EBADF] 70
 [EBUSY] 70
 [ECHILD] 70
 echo() 50
 [EDEADLK] 71
 [EDEADLOCK] 71
 [EDOM] 71
 [EDQUOT] 71
 [EEXIST] 71
 [EFAULT] 71
 [EFBIG] 71
 [EFTYPE] 71
 [EHOSTNOTFOUND] 72
 [EILSEQ] 72
 [EINPROGRESS] 72
 [EINTR] 72
 [EINVAL] 72
 [EIO] 72
 [EISDIR] 72
 [EMFILE] 72
 [EMLINK] 72
 [ENAMETOOLONG] 73

endwin() 51, 52

[ENFILE] 73

[ENODEV] 73

[ENOENT] 74

[ENOEXEC] 74

[ENOLCK] 74

[ENOMEM] 74

[ENOSPC] 74

[ENOSYS] 74

[ENOTBLK] 74

[ENOTDIR] 74

[ENOTEMPTY] 75

[ENOTTY] 75

environment variables

 _PSLIBPATH 62

 C89_COMPILER 20

 C89_ECHO 21

 C89_LINKER 20

 DISABLE_GETS_WARNING 28

 HOME 62

 INCLUDE 21

 LIB 21

 LOGNAME 62

 NCURSES_TRACE 54

 OPENNT_COMPILERDIR 9

 PATH 27, 62

 TZ 62

[ENXIO] 75

[EOPNOTSUPP] 75

[EPERM] 75

[EPIPE] 75

[EPROCLIM] 75

[ERANGE] 76

[EROFS] 76

errno variable 69

[ESPIPE] 76

[ESRCH] 75, 76

[ETOOMANYREFS] 76

[ETXTBUSY] 76

[EWOULDBLOCK] 70

[EXDEV] 76

exec() 30

F

fcntl() 26, 28

ffs() 37

fgets() 28

files

 /etc/passwd 39

 \$OPENNT_ROOT/usr/x11/lib/config/

 OpenNT.cf 66

 \$OPENNT_ROOT/pubs/GLPL.TXT 16

 /etc/groups 39

 displaying revision differences 14

 displaying revision history. *See* rlog utility

 fpathconf() 73

 Imakefile 14

 maximum length 73

 maximum size 71

 OPEN_MAX 72

 pathconf() 73

filter() 52

flock() 28

fpathconf() 27, 73

Free Software Foundation 18

FTP access, Softway Systems 11

function calls

 ioctl() 29

G

g++ utility 14, 18

gcc

 libraries 16

gcc utility 14, 18

getch() 50

getconf 73

getcwd() 26, 28

getenv() 30

getgrnam() 39

gethostid() 28

_getInstallPath() 41

getmaxyx() 55

getpwnam() 39

gets() 28

getstr() 55

getwd() 26, 28

global variables

 COLS 55

 errno 69

 LINES 55

GNU C 18–19

 Library Public License 18

gtty() 29

H

header files 15
 <limits.h> 27, 36
 <regex.h> 30
 <security.h> 40
 <string.h> 36
 <strings.h> 36
 <sys/stat.h> 26
 <sys/types.h> 27
 <termios.h> 29
 <utime.h> 36
 c89 search paths 21
 directory 20
 HOME environment variable
 for a service 62

I

imake utility 14
 Imakefile
 creating for Motif applications 67
 IMAXBEL flag 29
 INCLUDE environment variable 21
 index() 26, 37
 init_pair() 51
 initscr() 49, 52
 ioctl() 29, 30
 TIOCGETP 29
 TIOCGWINSZ 30
 TIOCSETP 29
 ioctl() 38

K

keypad() 50
 kill() 72
 killpg() 30
 ksh. *See* KornShell

L

ld utility 14
 leaveok() 49
 lex
 based on flex 14
 lex utility 14
 LIB environment variable 21
 libraries 15–16

 c89 search paths 21
 directory 20
 lex 21
 libc.a 15
 libcurses.a 15
 libg++.a 16
 libgcc.a 16
 libl.a 15
 libm.a 15
 liboldX.a 16
 libsocket.a 15
 libstdc++.a 16
 libtermcap.a 15
 libX11.a 16
 libXau.a 16
 libXaw.a 16
 libXdmcp.a 16
 libXext.a 16
 libXi.a 16
 libXmu.a 16
 libXtst.a 16
 liby.a 15
 license agreements 18
 psxdll.a 16
 yacc 21
 libraries provided 15–16
 link() 71
 LocalSystem (user login) 61
 lockf() 26, 28
 LOGNAME environment variable 62
 lseek() 76

M

macros
 _ALL_SOURCE 36
 PAGESIZE 36
 macros defined
 __STDC__ 21
 POSIX 21
 __OPENNT 21
 make utility 14
 \$(CC) macro 22
 \$(CFLAGS) macro 22
 conditionals 23
 extensions to POSIX.2 22
 including other makefiles 23
 makedepend utility 14

MAXPATHLEN 26
 MAXUPRC 26
 Microsoft Visual C/C++ 8–10, 13, 19–22
 and gcc 18
 CL.EXE 13
 header files 15
 linker compatibility with gcc 19
 Microsoft Visual C/C++ 5.0 21
 mkdirhier utility 14
 mkfifo() 26
 mkfifo() 28
 mknod() 26, 28
 mkstemp() 26
 mktemp() 26
 mmap() 36
 move() 49
 msgctl() 37
 msgget() 37
 msgrcv() 37
 msgsnd() 37
 mvaddch() 49
 mxmkmf utility 67, 68

N

NAME_MAX 73
 NCARGS 26
 NCURSES_TRACE 54
 newterm() 52
 newterm() 52
 newwin() 49
 NGROUPS 26
 NGROUPS_MAX 26
 nl() 49
 nm utility 14
 nocbreak() 55
 NOFILE 26
 noraw() 55

O

OPEN_MAX 26, 72
 __OPENNT
 isolating Interix-specific features 22
 __OPENNT macro 21
 OpenNT Release 2.0 10
 OPENNT_COMPILERDIR environment variable
 9

P

panels library 58
 passwd.pw_gecos member 40
 PATH environment variable 27, 62
 spaces in 9
 PATH_MAX 26, 73
 pathconf() 27, 73
 pathnames
 spaces in 8
 permissions
 Special Access permission. *See* ACE
 perror() 28
 POSIX 21
 POSIX subsystem
 entry points 16
 macros
 _POSIX_SOURCE 36
 _prefixInstallPath() 41
 printf() 49
 process environment 30
 programs
 psxrun.exe 61
 pseudo-terminals 38
 putenv() 30

R

rcs utility 14
 rcsdiff utility 14
 refresh() 53
 refresh() 45, 49
 regcomp() 30
 regerror() 30
 regexec() 30
 regular expressions 30
 rindex() 26
 rindex() 37
 ripofflines() 52
 rlog utility 14
 root user. *See* superuser

S

SA_RESTART flag 33
 scrollok() 49
 semctl() 37
 semget() 37
 semop() 37

- service control manager 61
- service utility 61
- services
 - coding 63
 - execution environment 62
- set_term() 52
- setuid programs 39
- setupterm() 53
- shmat() 37
- shmctl() 37
- shmdt() 37
- shmget() 37
- SIGABRT 32
- sigaction() 30, 31
- sigaddset() 30, 33
- SIGALRM 32
- sigblock() 30
- SIGCHLD 32
- SIGCLD 32
- SIGCONT 32
- sigdelset() 30, 33
- sigemptyset() 30, 33
- sigfillset() 30, 33
- SIGFPE 32
- sighold() 31
- SIGHUP 32
- sigignore() 31
- SIGILL 32
- SIGINT 32
- SIGIO 32
- sigismember() 30, 33
- SIGKILL 32
- signal() 26
- signal() 72
- Signals
 - BSD calls 30
- signals 30–33
 - [EINTR] 72
 - available signals 31–32
 - checking for a pending 33
 - data structures 32
 - SA_RESTART flag not supported 33
 - SIGABRT 32
 - SIGCHLD 32
 - SIGCLD 32
 - SIGIOT 32
 - signal mask 31
- sigpause() 30
- sigpending() 30, 33
- SIGPIPE 32
- sigprocmask() 30, 33
- SIGQUIT 32
- sigrelse() 31
- SIGSEGV 32
- sigset() 31
- sigset_t 32
- sigsetmask() 30
- SIGSTOP 32
- sigsuspend() 30
- SIGTERM 32, 61
- SIGTSTP 32
- SIGTTIN 32
- SIGTTOU 32
- SIGURG 32
- SIGUSR1 32
- SIGUSR2 32
- sigvec() 30
- SIGVTALRM 32
- SIGWINCH 32
- slk_init() 52
- sockets 37
- Softway Systems
 - addresses 11
 - communicating with 11
- <stdarg.h> 34, 34–35
- stdscr 45
- strcasecmp() 37
- strdup() 37
- strerror utility 14
 - description 14
- strerror() 28
- strip utility 14
- strncasecmp() 37
- strsep() 37
- struct passwd 40
- struct sigaction 33
- struct stat 26
- struct usersec 40
- struct winsize 30
- stty structure 29
- stty() 29
- subwin() 49, 55
- SVID
 - process environment 30
- symbolic links 26
- <sys/stat.h> 26
- sys_errlist 28
- sys_nerr 28
- sysconf() 27, 36
- sysctl() 27

sysinfo() 27

T

tcdrain() 30
 tcflow() 30
 tcflush() 30
 tcgetattr() 29
 tcgetpgrp() 30
 tcseendbreak() 30
 tcsetattr() 29
 tcsetpgrp() 30
 tempnam() 26
 TERM environment variable
 in curses 52, 54
 <term.h> 56
 termcap 37
 terminfo 37, 50
 termio structure 29
 termios structure 29
 tmpnam() 26
 touchwin() 50
 trace() 54
 ttytype[] 54
 TZ environment variable 62

U

uname() 28
 <unistd.h> 41
 update_panels() 58
 use_env() 52
 User Manager 40
 utilities 14
 ar 13
 as 13
 c89 13, 21, 66
 cc 13, 21, 66
 ci 13
 co 13
 cpp 14
 g++ 14, 18
 gcc 14, 18
 ident 14
 imake 14

ld 14
 lex 14
 makedepend 14
 mkdirhier 14
 mxmkmf 68
 nm 14
 rcs 14
 rcsdiff 14
 rlog 14
 service 61
 strerror 14
 strip 14
 xmkmf 14
 yacc 14
 utime() 26
 utime() 35
 <utime.h> 36
 utimes() 26
 utimes() 35

V

<varargs.h>. *See* <stdarg.h>
 VBELTIME flag 29
 VEOF flag 29
 VEOL flag 29
 VMIN flag 29
 VTIME flag 29

W

waddch() 46
 wait() 70
 waitpid() 70
 wbkgdset() 58
 wgetstr() 50
 WINDOW data type 45
 window size 30
 winsize structure 30
 wnoutrefresh() 53
 wnoutrefresh() 55
 wrefresh() 53
 wrefresh() 46
 wresize() 57
 write() 74
 wscanw() 50

X

xmkmf utility 14
xrdb utility 67
xtetris, building 67

Y

yacc utility 14