UnixWare Porting and Integration Guide

Chapter 1
Overview

Introduction to the UnixWare Porting and Integration Guide

The UnixWare Porting and Integration Guide is a resource for software developers who are planning to port existing software to UnixWare® from other UNIX® variants or to write new software for UnixWare. It provides a high level view of the UnixWare system for a software developer. It discusses interfaces for programming and administration of UnixWare, system tools available to the programmer, and extensive references to other UnixWare documentation that expand on concepts covered in this guide. This guide was written so that you have a single reference to start from when porting software to UnixWare or writing new software to run on UnixWare.

The integration sections of this guide includes principles, concepts, and feature overviews that will be important for those moving either applications or device drivers to UnixWare.

Intended Audience

The UnixWare Porting and Integration Guide is intended for software developers already familiar with one or more variants of the UNIX System, and who need information about the distinctions between UnixWare and other variants of the UNIX system.

This guide does not provide background information on other UNIX operating systems. It does, however, discuss specific interfaces, and how functionality provided in other operating systems is provided in UnixWare.

Organization of This Document

This guide discusses concepts in porting and integration and provides guidelines for writing software for UnixWare, but it is not a comprehensive technical reference. This guide contains numerous pointers to other documentation provided with UnixWare.

This guide is organized as follows:

1. Overview

   Provides a high−level view of the purpose and content of this guide.

2. UnixWare Programming Languages and Tools

   Discusses the UnixWare software development environment. It tells you about the facilities that exist in UnixWare to help you write and port software.

3. Developing Applications for UnixWare

   Describes supported UnixWare features, including ways you can write your programs to ensure forward compatibility in future releases of UnixWare as well as binary compatibility with other Intel −based UNIX variants. Also discusses supported user interfaces and preferred methods for providing system services in UnixWare.

4. Developing Device Drivers for UnixWare

Provides information about programming interfaces for device drivers in UnixWare. Also discusses a number of features that simplify writing and packaging device drivers, as well as simplifying user installation and administration.

5. Integrating Software with UnixWare

   Contains information to make your software look and feel like a part of the UnixWare system. This chapter discusses issues relevant to software packaging and desktop integration, which make your software manageable by the standard set of UnixWare administrative tools.

Notation Conventions

The following conventions are used in this guide:

- Computer input and output appear in `constant width` type. Substitutable values appear in **italic** type:

  ```
  $ cc file.c file.c file.c
  ```

  The dollar sign is the default system prompt for the ordinary user. There is an implied carriage return at the end of each command. When a command extends beyond the width of the page, the break is marked with a backslash and an indented second line:

  ```
  $ cc −L../archives −L../mylibs file1.c file2.c file3.c \
     file4.c −lfoo
  ```

  Of course, a command that extends beyond the width of your terminal screen will wrap around. You should use the backslash only if you enter the command exactly as shown.

- A number in parentheses following a command or function name refers to the section of the reference manuals where the command or function is described. For example, cc(1), means that the `cc` command is described in Section 1 of the reference manuals.

Related Documentation

Much of the information that is presented in this guide is treated in greater detail in the other reference manuals and programming guides distributed with the UnixWare Software Development Kit.

On−line documentation is available in UnixWare 2 and subsequent releases by means of the *Dyna*Text® document browser distributed with the system software. In addition to the document browser, there are three document sets that may be installed on your system: Application Server documentation, available for installation with the UnixWare Application Server; Personal Edition documentation, available for installation with the UnixWare Personal Edition; and the Software Development Kit documentation, available for installation with the UnixWare Software Development Kit. See the section *UnixWare User Documentation and Online Help* in the chapter *About This Handbook* in the book System Owner Handbook for more information about the on−line documentation provided with UnixWare.

UnixWare Application Programming Document Set

This document is part of a series of UnixWare programming guides which includes the following:

- Programming with System Calls and Libraries −−− Describes the Application Program Interface (API) for basic UnixWare operating system services provided by the kernel.
- Programming in Standard C and C++ −−− Discusses the UnixWare system programming environment and utilities, and provides details of the C and C++ languages, file formats, link editor, libraries, and tools.

- Software Development Tools ––– Describes various tools for developing and packaging application software, such as lex(1), yacc(1), as well as for modifying the sysadm(1M) interface.
- Network Programming Interfaces ––– Describes networking services such as the Transport Library Interface (TLI), the Remote Procedure Call (RPC), and the Network Selection facility.
- C++ Standard Components Programmer's Guide ––– Describes programming with a set of reusable C++ components.
- Motif Programmer's Guide ––– Describes programming with the Motif toolkit.
- Motif Style Guide ––– Describes user interface style guidelines when programming with the Motif toolkit.
- Windowing Korn Shell Programming ––– Describes programming with wksh(1), which combines the simplicity and quickness of programming in the Korn Shell ksh(1) with the graphical interface of Motif.
- Character User Interface Programming ––– Describes the older SVR4 interfaces for programming character terminal displays; `curses` and terminfo(4), ETI (Extended Terminal Interface), and FMLI (Form and Menu Language Interpreter) are covered.

UnixWare IHV Development Kit Documentation

This document is included with the UnixWare Software Development Kit Documentation Set. It includes the following:

- Device Driver Programming ––– A comprehensive guide on how to write, configure, and package device drivers
- Portable Device Interface (PDI) ––– A programming guide on writing device drivers to use the Portable Device Interface (PDI), a framework that is designed to organize, simplify, and standardize the way device drivers operate.
- STREAMS Modules and Drivers ––– A programming guide on writing STREAMS drivers.
- Device Driver Reference ––– Previously known as the DDI/DKI, this reference manual documents all the routines and structures that make up the Device Driver Interface/Driver–Kernel Interface, as well as the routines and structures specific to PDI and STREAMS.

Reference Manuals

The reference manual set contains manual pages which describe features of the UnixWare operating system. This documentation is a companion set to the UnixWare programming guides. The reference manual set includes the following:

- Command Reference
- System Files and Devices Reference
- Operating System API Reference
- Motif Reference Manual
- C++ Standard Components Programmer's Reference
- Device Driver Reference
- NetWare Library Reference for C: Client Functions

When software components are mentioned in this guide, a reference to the manual section where the software component is formally described is included in parentheses: *`component_name`*(*`section_number`*). The numbered sections are located in the following manuals:

Section (1)
: Command Reference

Section (2)
: Operating System API Reference

Section (3)
: Operating System API Reference, or the Motif Reference Manual,

Sections (4), (5), (7)
: System Files and Devices Reference

Manual pages are referred to with the command, system call, function, or device name showing first in `Courier` font, followed by the section number appearing in parenthesis in the normal font. For example, the Executable and Linking Format Library (ELF) manual page appears as elf(3E).

Porting and Integration Concepts

Porting Concepts

Porting software is the process of making software that was written for one operating environment work in another operating environment. For example, you may have written a program for the SCO Xenix system, and now want to make the same program run on UnixWare. The software you start with is the main factor in determining the difficulty of the task of porting it to UnixWare.

The task of porting software from one operating system to another ranges in complexity from simply installing the binary on the new system, to recompiling on the new system, or to making major changes in the design of the software. If you are planning on writing software for a larger market than just one system, it's a good idea to plan for portability.

In addition to providing information on how to port your software to UnixWare from some of the other UNIX systems, this guide discusses some of the issues involved in writing portable code. It's possible to write software that can run as–is, in binary form, on different operating systems on one hardware platform. This guide also discusses how you can write software that takes advantage of industry standard interfaces that let you cover a larger market with one binary product.

Integration Concepts

Once a package has been ported to UnixWare it will run on the system. The next step will be making it work with all the other pieces of software that make up the UnixWare system. This is the integration process.

When porting software to UnixWare or writing new software, it's important to keep in mind the issues that make your software an integral part of the system you're writing for. When a user installs your software on their system, it's helpful if your software conforms to the conventions of that system. This allows your users to manage and interact with your software in a way that's consistent with the rest of their system.

Software packages are managed objects in the UnixWare environment. UnixWare users have at their disposal a number of commands for installing, uninstalling, and finding information about software installed on their systems. When users receive your software and it's in the proper format, familiar system tools can be used to install it.

Additionally, UnixWare comes with a graphical user interface; the UnixWare Desktop. The Desktop consists of a number of windows displaying icons representing all the files, folders, applications, and utilities that come with UnixWare. See the section *What's on Your Desktop?* in the chapter *Introduction* in the book Desktop User Handbook for an introduction to the UnixWare Desktop.

With little effort, you can have your software available to be executed and managed from the UnixWare Desktop. This guide contains discussions of the issues you need to keep in mind when writing and packaging your software so that it can be an integral part of the UnixWare systems on which it is installed.

Chapter 2
UnixWare Programming Languages and Tools

Introduction to UnixWare Programming Languages and Tools

This chapter introduces tools that are available to programmers writing software on UnixWare. As tools and resources are described, pointers are provided to more detailed information in other parts of the UnixWare programming document set.

The UnixWare Software Development Kit provides developers with a wealth of tools and utilities for writing, debugging, packaging, and maintaining software. Complete information on these tools are available in the programming series guides and reference manuals distributed with the UnixWare Software Development Kit. This chapter highlights the programming environments and tools of interest to software developers writing applications and system software for UnixWare.

C Compilation System

The UnixWare Software Developer's Kit contains an ANSI–conforming C compilation system, as well as a number of debuggers, profilers, and other programmer productivity tools. Information about the C Compilation system may be found in Programming in Standard C and C++

In addition, there are a number of other useful programming guides and reference manuals provided with the UnixWare Software Developer's Kit:

- Software Development Tools. This programming guide is about UNIX system tools used to create, maintain, and extend C programs.
- Programming with System Calls and Libraries. This programming guide concentrates on how to use the system services provided by the UnixWare operating system kernel. It is designed to give you information about application programming in a UnixWare environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming by concentrating on the other elements that are part of getting application programs into operation.
- Operating System API Reference. This reference manual describes the C language interface used by application programs to access the operating system services of UnixWare 2. The UnixWare operating system application programming interface (API) described in this reference manual includes system calls and C library functions.

C++ Compilation System

The organization of the C++ compilation system is similar to that of the C compilation system. It consists of a preprocessor, compiler, assembler, and link editor. There is also a C++ name filter (demangler), a C++ patch utility, and a C++ prelink phase. The CC command invokes each of these components automatically, unless otherwise directed, to create an executable C++ program from the source code. For more details about the structure of the C++ compilation system on UnixWare, see the section Components of the C++ Compilation System in the chapter *C and C++ Compilation System* in the book *Programming in Standard C and C++*

In addition, there are a number of other useful references provided with the UnixWare Software Developer's Kit:

- Programming with the C++ Standard Libraries. This programming guide describes the C++ class libraries provided with the C++ compiler for complex arithmetic and I/O.
- C++ Standard Components Programmer's Guide. This programming guide describes the standard software components that are a part of the UnixWare Software Developer's Kit, and the collection of development tools that support component−oriented application development.
- C++ Standard Components Programmer's Reference. This reference manual describes the use of the C++ Standard Components and provides manual pages for each tool and component.
- Debugging and Analyzing C and C++ Programs This guide covers using the Enhanced Debugger to identify problems in C++ programs.

debug −−− the Enhanced Debugger

debug(1) is a symbolic, source−level debugger with many capabilities, including an easy−to−understand command language, control over multiple processes, and evaluation of complex expressions. It facilitates the finding of errors in user programs by allowing the user to control the execution of a program and examine its state. Live programs can be executed one source statement or machine instruction at a time, or can be instructed to run until some event occurs. Program variables and the processor registers may be examined or modified, and the user can request a trace back of active functions, disassembly of a portion of the program's executable code, or a raw dump of any area of the program's memory.

`debug` supports single and multiprocess applications and processes that consist of a single thread of control or of multiple threads. Most of the debugger commands accept a list of processes and threads to which the specified actions are applied. The user can set a breakpoint in a single thread and request that a list of threads and processes stop when that breakpoint is encountered. Users can also request interception of signals and system calls on a per−thread basis, and can request per−thread stack traces and register dumps.

For more information on the command−line interface of `debug`, see the chapter Using the Command Line Interface of debug in the book *Debugging and Analyzing C and C++ Programs.*

The graphical interface of `debug` displays its information in several different windows. Each window is composed of one or more panes, each pane providing a different type of information.

The debugger's informational panes are arranged in a set of windows, as described by the configuration provided by the user. Each window set contains at least one instance of each of the debugger's nine panes (except for the Status pane, which is not required). If no user configuration is provided, the debugger arranges its panes into a set of six windows.

Each window set controls a subset of the processes and threads being debugged, and each thread or process belongs to exactly one window set. Multiple window sets can be created in any one debugging session. If you are debugging only one single−threaded process, you will not need more than one window set, but if you are debugging multiple threads or processes, you may wish to debug each thread or process in a separate window set.

For more information on the graphical interface of `debug`, see the chapter Using the Graphical Interface of debug in the book *Debugging and Analyzing C and C++ Programs*

prof, lprof, and fprof −−− The Profiling Utilities

Profilers are tools that analyze the dynamic behavior of your program: how fast, how often, and when the parts of its code are executed. First, the profilers interpret data that have been collected about that behavior at run time; then they display the interpreted data in a format you choose from a number of options.

Three complementary utilities are available for profiling C and C++ source programs:

- prof(1) is a time profiler. It reports the amount of time and the percentage of time that was spent executing the parts of a program. It also reports the number of calls to each function and the average execution time of the calls. For complete information on using prof, see the section Using prof in the chapter *Analyzing Run−Time Behavior* in the book *Debugging and Analyzing C and C++ Programs.*

- lprof(1) is a line−by−line frequency profiler. It reports whether and how many times each line of source code was executed. It lets you identify the unexecuted and most frequently executed parts of your code. For complete information on using lprof, see the section Using lprof in the chapter *Analyzing Run−Time Behavior* in the book *Debugging and Analyzing C and C++ Programs.*

- fprof(1) is a flow and time profiler. Currently, there is no support for C++. It reports when each function was called and when the function exited. This information can be used for performance analysis or **Locality of Reference Tuning**.

  Locality of Reference Tuning is a process by which functions are relocated within an object such that functions that are called at similar times are placed in the same area of the object. This is important because it reduces page−fault activity and provides better cache usage.

  On machines where memory is short, this can be the most important action that can be taken to speed up a program. For example, by tuning the locations of functions within the X−Windows libraries, we sped up start−up time for the UnixWare desktop by 300%.

  Flow profiling also has statistical support to show how much total time (and number of calls) was spent executing the parts of a program. This statistical feature also gives caller/callee information and allows the user to select the period of time that should be used in the statistical analysis. It also allows you to query the logs to see the actual flow of control. For examples of `fprof` output, see the section Using fprof in the chapter *Analyzing Run−Time Behavior* in the book *Debugging and Analyzing C and C++ Programs*, and the section A Sample Session Using Flow Profiling in the chapter *Analyzing Run−Time Behavior* in the book *Debugging and Analyzing C and C++ Programs*.

For detailed information about what benefits you can get from using the profiling tools in the UnixWare Software Developers Kit, as well as how to use them, see the section Profiling Utilities in the chapter *Analyzing Run−Time Behavior* in the book *Debugging and Analyzing C and C++ Programs.*

kdb −−− the Kernel Debugger

The UnixWare SDK provides an extremely useful tool for debugging device drivers: the kernel debugger (also known as kdb(1M)). Refer to the kdb(1M) manual page in the

`kdb` can set breakpoints, display kernel stack traces and various kernel structures, and modify the contents of memory, I/O, and registers. The debugger supports basic arithmetic operations, conditional execution, variables, and macros. `kdb` does conversions from a kernel symbol name to its virtual address, from a virtual address to the value at that address, and from a virtual address to the name of the nearest kernel symbol. You have a choice of different numeric bases, address spaces, and operand sizes.

For more information on using `kdb`, see the section Kernel Debugger in the chapter *Driver Testing and Debugging* in the book *Device Driver Programming.*

IHV Development Kit

The Novell UnixWare Independent Hardware Vendor (IHV) Development Kit is designed to help you to write and package your own UnixWare device drivers. The IHV Development Kit Roadmap gives you a tour of the contents of the UnixWare IHV Development Kit and provides a starting point for answering your questions about writing device drivers. Much as this UnixWare Porting and Integration Guide guide is designed as a single reference covering a broad range of issues in the area of writing software; the IHV Development Kit Roadmap is a single reference covering a range of issues in the area of writing device drivers for UnixWare.

Software Packaging Utilities

UnixWare provides a variety of tools for managing software. Foremost among these tools are pkgadd(1M) and pkgrm(1M) used, respectively, for adding and removing software packages. Software packaged so that it can be managed by these utilities is easier to maintain, more flexible in its installation, and more convenient for users.

In the UnixWare Software Developer's Kit, the pkgmk(1) packaging tool is provided to help automate package creation. It gathers the components of a package on the development machine, copies them onto the installation medium, and places them into a structure that pkgadd(1M) recognizes.

See the chapter Packaging Your Software Applications in the book *Software Development Tools* for detailed information on packaging your software.

Chapter 3
Developing Applications for UnixWare

Introduction to Developing Applications for UnixWare

UnixWare provides a rich programming environment, with a high level of support for standards (such as the ABI, XPG4, and XTI) and functionality that provides convenience for programmers. This enables you to write programs that combine a high degree of portability to other systems with the ability to take advantage of advanced UnixWare features.

This chapter provides overviews of a number of issues that may not be obvious to programmers who are familiar with other UNIX−based operating systems. It describes UnixWare features of interest to application programmers, and provides references into other programming guides that are distributed as part of UnixWare's online documentation set.

You are encouraged to browse through the books in the Programming (UnixWare SDK Guides) collection for more information and for additional ideas about how to take advantage of UnixWare features.

ABI Conformance

You should seriously consider the benefits of building your products in conformance with the binary compatibility specification that matches your intended market. For UnixWare and other UNIX System V−derived operating systems on the Intel Architecture, this specification can be found in the following documents:

- *System V Application Binary Interface (ABI)* Discusses ABI issues that are generic to all hardware platforms.
- *Intel Processor Supplement* Covers issues that are specific to the Intel platform.
- *ABI+ Specification for Intel Architectures* An extension of the ABI with additional Intel specifics.

These documents form a set, where each subsequent document in the list adds to information provided by the others.

These specifications include information for both system vendors and for application developers. For system developers, they specify the APIs that must be provided, and the formats of binary files and data that are necessary for binary compatibility. For application developers, they list the APIs guaranteed to be present on conforming systems, and the way applications should be built and packaged so that their binaries will install and run on all conforming systems.

The benefit is that the same binary distribution of your product run on systems from many vendors, rather than having to be ported and compiled separately for each system. You can sell each shrink−wrapped unit into a significantly larger market than would otherwise be the case.

Transport Independent Network Programming

The X/Open Transport Interface (XTI)

One of the advantages of UnixWare is the availability of XTI, the X/Open Transport Interface. Unlike the older sockets mechanism, XTI allows programmers to create applications that will run without modification on a variety of networks. XTI does this by providing a standard means of gaining direct access to transport services in a transport−independent way.

This poses an advantage because networking programs written to use XTI services can take advantage of multiple networking protocols without change. In UnixWare 2, for example, there is support for both the TCP/IP and IPX/SPX protocols. Writing code that uses XTI services allows your programs to use either protocol, and even determine which systems provide which services using which protocols. Examples of this in UnixWare are:

- tsad(1), the daemon for Storage Management Services (SMS) Target Service Agent (TSA) for UnixWare
- pkgcopy(1M), pkginstall(1M), and pkglist(1M), the Network Installation Utilities.

Both of these facilities use the same executable to communicate over both IPX/SPX and TCP/IP, and they are able to determine the presence of systems that provide the appropriate services on these networks.

The UNIX system Transport Interface was modeled after the industry standard ISO Transport Service Definition (ISO 8072). The resulting interface was called the Transport Level Interface (TLI) library. The TLI library functions have been modified to comply with the X/Open Transport Interface (XTI) specification. Both the TLI and the XTI library interfaces are available as part of the UNIX system Transport Interface. Applications compiled on earlier releases of UnixWare will get TLI semantics. However, new applications will be compiled so that they call the new XTI library interfaces.

The Transport Interface is implemented as a user library using the STREAMS input/output mechanism. Therefore, many services available to STREAMS applications are also available to users of the Transport Interface. For detailed descriptions of these services, see the chapter X/Open Transport Interface (XTI) in the book *Network Programming Interfaces*. For detailed information about STREAMS , see the STREAMS Modules and Driversguide.

Sockets Migration and Sockets−to−XTI Conversion

The sockets interface was first introduced as part of the Berkeley 4.2 Software Distribution. A significant application base has been written using this interface. Socket support was added to UNIX System V Release 4 as part of the BSD/System V unification. The sockets interface is supported by the shared library `libsocket.so` in UnixWare 2. For detailed descriptions of socket services provided in UnixWare 2, see the chapter The Sockets Interface in the book *Network Programming Interfaces*.

Although existing sockets applications can be rewritten for XTI relatively easily, such ports are not necessary for sockets applications that are run only over TCP/IP or UDP/IP networks. However, XTI is the preferred programming interface for accessing transport services and it is recommended that programmers writing new applications for UnixWare 2 use XTI.

There are areas in which there is no direct correspondence between XTI and sockets routines. Also, differences exist between UnixWare sockets and BSD sockets. You must be aware of these differences before moving BSD sockets applications to UnixWare 2. For detailed descriptions of these issues, please see the section Sockets Migration and Sockets−to−XTI Conversion in the chapter *The Sockets Interface* in the book *Network Programming Interfaces*.

Writing Server Code

Overview

When writing server code for a networked application in UnixWare, there are a couple of options available for ensuring that the server responds to client requests for network services. One option is to write your own network listener process. The other option is to use the system daemons inetd(1M) and nwnetd(1M) for this purpose. This section provides some information on the advantages and disadvantages of each approach, as well as some pointers for using each approach.

inetd and nwnetd

By far, the simplest way to provide a network service is to use the system daemons `inetd` and `nwnetd`, which listen for connections on the TCP/IP and IPX/SPX ports, respectively, of the services that their configuration files, inetd.conf(4) and nwnetd.conf(4) specify. When a connection is found, these daemons invoke the server daemon specified by that configuration file for the service requested.

When finished with the server process, `inetd` and `nwnetd` continue to listen on the port. The term `finished` refers to one of two situations, which are configurable on the system; `inetd` and `nwnetd` can be configured to accept new connections either after the server process is connected, or after the server process has exited. This lets an application tailor its environment so that multiple connections to a service may be made concurrently, or so that only one connection is active at once.

Rather than having several daemon processes with sparsely distributed requests each running concurrently, `inetd` and `nwnetd` reduce the load on the system by invoking server processes only as they are needed. In addition, they also simplify the implementation of server processes. Any process started by `inetd` or `nwnetd` is started with its file descriptors 0, 1, and 2 (`stdin`, `stdout`, and `stderr`) corresponding to the network connection started by these daemons. In this way, any server process may be written as though it were a local, non−network−aware process that communicates with clients via normal UNIX file descriptors, with no need for network semantics.

Writing Your Own Listener

There are situations where it is more appropriate to write your own listener process for a network service. In the simplest cases, this is not necessary and leads to the writing of code that already exists in `inetd` and `nwnetd`, but there are cases where an application needs more control over connections than are provided by these system daemons.

Following are some of the situations where it is desirable to write your own listener process:

- You need to have fine control of networking parameters from your server.
- You need to add network−aware code to your server process, such as a session layer that can provide extra reliability over slow or unreliable links beyond what the underlying networking protocols guarantee.
- You need to control the number of concurrent server processes on the server machine. It’s possible in this situation to have your listener pass client connections to a queue where they’re held until one of the possible server processes becomes free.
- You want a custom listener to manage multiple connections and pass information between them. This covers not only controlling the number of servers, but things like networked multiuser systems where clients are talking to each other.

Advertising Your Server

UnixWare 2 provides a set of APIs for advertising a server via the Service Advertising Protocol (SAP) over IPX. This set of APIs provides both advertising and service query capability to the Server or Client programmer. For more information, see the section NetWare Transport Programming in the chapter *Developing Applications for UnixWare* in the book *UnixWare Porting and Integration Guide* later in this document.

Programming Examples

Following is a useful example of how to program using XTI services and the `netdir` facility, which is a way for your software to select an appropriate networking protocol. Additional programming examples may be found in the section Some Examples in the chapter *X/Open Transport Interface (XTI)* in the book *Network Programming Interfaces.*

Example Server Code

The program `server`, shown in the following example, is run with two arguments: the protocol that it is to use, and a service name to implement. The protocol should be either `tcp` or `SPX`; the service name can be any service that is specified in the `/etc/services` file but is not currently listened for by a specific daemon. Frequently the `daytime` service works for this purpose.

When the program is executed, it uses the `netdir_getbyname` function [see netdir(3N)] to determine the correct address to listen to for the service and protocol specified. No other protocol specific statements are necessary.

This program may be compiled with the command line:

```
cc -o server server.c -lnsl
```

The source code in server.c is as follows:

```
#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>
#include <netconfig.h>
#include <netdir.h>
#include <poll.h>
struct netconfig *netEntry ;
static char  *Data = "Well hi there, Mr World!" ;
static int  timeout = 1000 ;
extern int  t_errno ;
/***
 *
 *  usage: server <protocol> <service name>
 *  eg: server spx daytime
 *
 ***/
main(int argc, char **argv)
    {
    struct t_bind *tbind, *tbound ;
    struct nd_hostserv hostService ;
    struct nd_addrlist *serviceData ;
    struct netbuf *serviceAddress ;
    struct t_call *callData ;
    struct pollfd pollFds[2] ;
    int   listenFd, acceptFd ;
    int   childpid ;
    /* get network configuration entry for the protocol specified */
    if ( (netEntry = getnetconfigent(argv[1])) == NULL )
 {
 perror("network configuration entry not found") ;
```

```
exit(1) ;
}
   /* get address to bind to */
   hostService.h_host = HOST_SELF ;
   hostService.h_serv = argv[2] ;
   printf("Getting address for service %s under %s\n", argv[2], argv[1]) ;
   if ( netdir_getbyname(netEntry, &hostService, &serviceData) != 0 )
{
perror("cannot get address") ;
exit(1) ;
}
   /* point serviceAddress to the address out of the serviceData structure */
   serviceAddress = serviceData->n_addrs ;
   printf("\t%s: ",  taddr2uaddr(netEntry, serviceAddress)) ;
   /* open the indicated device for this protocol */
   if ( (listenFd = t_open(netEntry->nc_device, O_RDWR, (struct t_info *)NULL)) < 0 )
{
perror("cannot t_open") ;
exit(1) ;
}
   /* allocate tbind structures for this protocol */
   if ( (tbind = (struct t_bind *) t_alloc(listenFd, T_BIND, T_ALL)) == NULL )
{
perror("cannot t_alloc tbind") ;
exit(1) ;
}
   if ( (tbound = (struct t_bind *) t_alloc(listenFd, T_BIND, T_ALL)) == NULL )
{
perror("cannot t_alloc tbound") ;
exit(1) ;
}
   /* copy the address from serviceAddress into the tbind address above */
   tbind->qlen = 5 ;
   tbind->addr.len = serviceAddress->len ;
   (void) memcpy(tbind->addr.buf, serviceAddress->buf, serviceAddress->len) ;
   /* bind to that address, return bound address in tbound */
   if ( t_bind(listenFd, tbind, tbound) != 0 )
{
perror("cannot t_bind") ;
exit(1) ;
}
   /* allocate a t_call structure for this protocol */
   if ( (callData = (struct t_call *) t_alloc(listenFd, T_CALL, T_ALL)) == NULL )
{
perror("cannot t_alloc callData") ;
exit(1) ;
}
   while(1)
{
/* listen for connection */
if ( t_listen(listenFd, callData) < 0 )
    {
    t_error("error from t_listen") ;
    exit(1) ;
```

```
    }
/* got a call, open an accepting port */
if ( (acceptFd = t_open(netEntry->nc_device, O_RDWR, (struct t_info *)NULL)) < 0 )
    {
    t_error("cannot open acceptFD") ;
    exit(1) ;
    }
/* bind accepting port */
if ( t_bind(acceptFd, NULL, NULL) != 0 )
    {
    t_error("Cannot t_bind") ;
    exit(1) ;
    }
/* accept the connection on that port */
if ( t_accept(listenFd, acceptFd, callData) != 0 )
    {
    t_error("cannot t_accept") ;
    exit(1) ;
    }
/* send the data */
if ( t_snd(acceptFd, Data, strlen(Data), 0) < 0 )
    {
    t_error("error from t_snd") ;
    if ( t_errno == TLOOK )
 printf ("tlook = %d\n", t_look(acceptFd)) ;
    }
/* wait a short while for the client to go away */
pollFds[0].fd = acceptFd ;
pollFds[0].events = POLLHUP ;
pollFds[0].revents = 0 ;
(void) poll(pollFds, 1, timeout) ;
/* close down */
t_close(acceptFd) ;
}
   }
```

Example Client Code

The `prod` program, shown in the following examples, is a protocol independent client. It is also run with two arguments: a machine name and a service name. When run, this program uses the getnetpath(3N) function to loop through every protocol that the host machine knows about, and check each one to see if that protocol can be used to generate an address to the specified machine and service. If a valid address is found, and the protocol indicated is connection oriented, then `prod` will attempt to contact the address to see if it has anything to say.

If you ask `prod` to contact the same machine and service that `server` is running on, then the server program will respond with a simple `hello world`.

This program may be compiled with the command line:

```
cc -o prod prod.c -lnsl
```

The source code in `prod.c` is as follows:

```
/***
 *
 *  name prod.c - poke at a service
```

```
 *
 *  synopsis prod <machine> <service>
 *
 ***/
#include <tiuser.h>
#include <netconfig.h>
#include <netdir.h>
#include <stdio.h>
#include <nl_types.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>
#include <stropts.h>
#include <poll.h>
#define BUFSIZE 256
#define TIMEOUT 20
int  InterruptSignal ;
extern int t_errno ;
SignalTrapper( int Signal )
    {
    InterruptSignal = Signal ;
    }
main(argc, argv)
int argc ;
char **argv ;
    {
    struct sigaction sigAct ;
    char  *hostName, *serviceName ;
    char  buffer[BUFSIZE], ctlBuffer[BUFSIZE] ;
    void  *netHandle ;
    struct netconfig *netEntry ;
    struct nd_hostserv hostService ;
    struct nd_addrlist *serviceData ;
    struct t_call *callData ;
    struct netbuf *serviceAddress ;
    struct pollfd pollFds[2] ;
    unsigned long numFds ;
    struct strbuf ctlBuf, dataBuf ;
    int   flags ;
    int   loop ;
    int   rc ;
    int   fd ;
    int   timeout = TIMEOUT ;
    if ( argc != 3 )
 {
 fprintf(stderr, "usage: %s <machine name> <service>\n", argv[0]) ;
 exit(1) ;
 }
    /* set up alarm trap */
    sigemptyset(&sigAct.sa_mask) ;
    sigAct.sa_flags = 0 ;
    sigAct.sa_handler = (void (*)()) SignalTrapper ;
    sigaction(SIGALRM, &sigAct, NULL) ;
    /*
```

```
     * NOTE: a more modern approach would be to use cs_connect for this
     */
    hostService.h_host = argv[1] ;
    hostService.h_serv = argv[2] ;
    /* check for magic host names */
    if ( strcmp(hostService.h_host, "-a") == NULL )
hostService.h_host = HOST_ANY ;
    else if ( strcmp(hostService.h_host, "-s") == NULL )
hostService.h_host = HOST_SELF ;
    else if ( strcmp(hostService.h_host, "-b") == NULL )
hostService.h_host = HOST_BROADCAST ;
    if ((netHandle = setnetpath()) == NULL)
{
nc_perror(argv[0]);
exit(1);
}
    while ((netEntry = getnetpath(netHandle)) != NULL)
{
/* netconfigp now describes a transport provider */
rc = netdir_getbyname(netEntry, &hostService, &serviceData) ;
if ( rc == 0 )
    {
    printf("Protocol %s returned %d addresses.\n", netEntry->nc_netid, serviceData->n_cnt) ;
    serviceAddress = serviceData->n_addrs ;
    for (loop=0; loop<serviceData->n_cnt; loop++)
 {
 printf("\t%s: ",  taddr2uaddr(netEntry, serviceAddress)) ;
 fflush(stdout) ;
 /* attempt to connect to service */
 if ( (fd = t_open(netEntry->nc_device, O_RDWR, (struct t_info *)NULL)) < 0 )
     printf("cannot open\n") ;
 else if ( netEntry->nc_semantics == NC_TPI_CLTS )
     {
     /* connectionless protocol found */
     printf("connectionless\n") ;
     }
 else
     {
     /* connection oriented protocol: try to connect */
     if ( (rc = t_bind(fd, (struct t_bind *)NULL, (struct t_bind *)NULL)) < 0 )
  {
  fprintf(stderr, "\nCannot bind, t_errno = %d\n", t_errno) ;
  exit(1) ;
  }
     else if ( (callData = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR)) == NULL )
  {
  fprintf(stderr, "\nCannot allocate callData, t_errno = %d\n", t_errno) ;
  exit(1) ;
  }
     callData->addr.len = serviceAddress->len ;
     (void) memcpy(callData->addr.buf, serviceAddress->buf, callData->addr.len) ;
     InterruptSignal = 0 ;
     alarm(timeout) ;
     rc = t_connect(fd, callData, (struct t_call *)NULL) ;
```

```
   alarm(0) ;
   if ( rc != 0 )
{
/* connection attempt failed */
if ( InterruptSignal == SIGALRM )
    printf("failed: timed out\n") ;
else if ( t_errno == TLOOK )
    {
    rc = t_look(fd) ;
    switch(rc)
 {
 case T_DISCONNECT :
     printf ("failed: hung up on me!\n") ;
     break ;
 default:
     printf("failed: reason code %d", rc) ;
     t_error("") ;
     break ;
 }
    }
else
    {
    printf("Connection failed.\n") ;
    printf("%d - %d\n", errno, t_errno) ;
    t_error("") ;
    }
}
   else
{
printf("success\n") ;
pollFds[0].fd = fd ;
pollFds[0].events = POLLIN | POLLPRI ;
pollFds[0].revents = 0 ;
numFds = 1 ;
if ( (rc = poll(pollFds, numFds, timeout*100)) < 0 )
    perror("poll failed") ;
else if ( rc > 0 )
    {
    /* it said something, lets look at it */
    flags = 0 ;
    ctlBuf.buf = ctlBuffer ;
    ctlBuf.maxlen = BUFSIZE ;
    dataBuf.buf = buffer ;
    dataBuf.maxlen = BUFSIZE ;

    rc = getmsg(fd, &ctlBuf, &dataBuf, &flags) ;
    if ( rc < 0 )
 {
 perror("bad read") ;
 break ;
 }
    else if ( dataBuf.len > 0 )
 printf("\tRead %d characters: [%s]\n", dataBuf.len, dataBuf.buf) ;
    }
```

```
    }
       t_close(fd) ;
       t_free((char *) callData, T_CALL) ;
       }
   serviceAddress++ ;
   }
      netdir_free(serviceData, ND_ADDRLIST) ;
      }
  else
      fprintf(stderr, "service not found for protocol %s: %s\n", netEntry->nc_netid, netdir_sperror()) ;
  }

     endnetpath(netHandle);
     }
```

NetWare Transport Programming

Introduction to NetWare Transport Programming

UnixWare provides a programming interface to the communication protocols in the NetWare Protocol Stack. The complete stack consists of daemons, various drivers, and protocol–specific modules and can be run on the host system independent of the NetWare service modules. If you are just starting out, it is a good idea to read the chapter UnixWare: Getting Started in the book *NetWare Developer's Guide for C*.

A number of protocols are available in the NetWare Protocol Stack. Each provides specific services, and many have a specific programming interface. The available protocols are:

- Internetwork Packet Exchange (IPX) Protocol

  NetWare's connectionless datagram service protocol.

- Routing Information Protocol (RIP)

  NetWare's distance vector protocol that is responsible for maintaining a list of distances to each destination network on a network.

- Sequenced Packet Exchange (SPX) Protocol

  NetWare's connection–oriented, reliable, sequenced transport protocol.

- Service Advertising Protocol (SAP)

  NetWare's protocol for file servers, print servers, and gateway servers, to register their services and addresses in a Server Information Table and have these services advertised across a network.

The following sections present an overview of each of the available NPS protocols.

Internetwork Packet Exchange (IPX) Protocol

IPX is a connectionless, datagram service protocol that does not require an acknowledgment for each packet sent.

No connection between the stations is established when a client uses this transport protocol to communicate with other clients or servers. Because IPX is a datagram service, each packet is routed individually to its destination on a network or internetwork.

For more information on how the IPX protocol works and how to program to the IPX programming interface, see the chapter Internetwork Packet Exchange (IPX) Protocol in the book *NetWare Transports* and the chapter IPX Direct Interface in the book *NetWare Transports*.

Routing Information Protocol (RIP)

RIP is a distance vector protocol that is responsible for maintaining a list of distances to each destination network on an internetwork.

RIP has no application programming interface. In UnixWare, routing is implemented as a LAN router within the IPX driver for servers and dedicated routers. The client configuration of the IPX driver does not support routing. For more information on how the RIP protocol works, see the chapter Routing Information Protocol (RIP) in the book *NetWare Transports*.

Sequenced Packet Exchange (SPX) Protocol

SPX is a connection–oriented, reliable, sequenced transport protocol. SPX provides a packet–level service, while enhanced SPXII provides a message–level service.

SPX guarantees packet delivery to the destination endpoint and notifies the user if any errors occur during data transmission. Upon encountering a data transmission error, SPX retries a given number of times before closing the connection and notifying the connection user. SPX also notifies the user if a disconnection indication is received from the remote connection endpoint.

For more information on how the SPX protocol works and how to program to the SPX programming interface, see the chapter Sequenced Packet Exchange (SPX) Protocol in the book *NetWare Transports*.

Service Advertising Protocol (SAP)

SAP provides a way for service nodes, such as file servers, print servers, and gateway servers, to register their services and addresses in a Server Information Table and have these services advertised across an internetwork.

SAP is a distance vector protocol where the destinations are text strings. End nodes can listen for SAP or make queries to get a specific service or all services.

SAP provides dynamic registration of services. Periodic broadcast packets provide the service information and addresses for the Server Information Table. When a server goes down or fails to send a periodic broadcast packet, that service is removed from the table.

For more information on how the SAP protocol works and how to program to the SAP programming interface, see the chapter Service Advertising Protocol (SAP) in the book *NetWare Transports* and the chapter SAP Library in the book *NetWare Transports*.

NetWare Client APIs for C

This set of libraries allows the programmer to write applications to interact with NetWare servers and services. These

APIs utilize the NetWare Unix Client (NUC) requester to connect to and communicate with servers.

For more information, see the book NetWare Library Reference for C: Client Functions which describes the NetWare APIs for bindery services and NetWare Directory Services (NDS); and the book NetWare Developer's Guide for C which gives an overview of the available NetWare services.

NUC Utilities

There are several utilities available which provide basic NetWare server connection functionality to the user. These are useful for removing the programming obstacles related to NetWare connection and security protocols. For more information on these facilities, see the reference manual pages `nwlogin(1), nwlogout(1), nwchecksums(1M), nwsignatures(1M), nwprimserver(1), nwwhoami(1),` and `setpass(1).`

User Interfaces

In application programming, the presentation of the software is always an issue. The software should be easy and fairly intuitive to use, and software that has a good `look and feel` makes a lasting impression on its users. UnixWare provides a variety of tools and programming interfaces for enabling user interfaces. These include tools such as menu(1), a menu and form interface generator, and wksh(1), an extended version of ksh(1) that provides the ability to program in Motif via shell functions. Also provided are lower−level programming libraries such as curses and ETI, the Extended Terminal Interface for writing forms and menus, and libraries for interfacing with the X Window system and the UnixWare Desktop.

A number of these interfaces are briefly described in the following sections; more information about each will be found in other documentation referred to in this section.

Character User Interface Tools

UnixWare provides tools for writing character user interfaces for software. These tools are invoked from a command level, not from within application binary code. Application programming interfaces for writing character user interfaces are discussed in the section Character User Interface APIs later in this chapter.

Form and Menu Language Interpreter

FMLI is a high−level programmer interface for creating menus, forms, and text frames that enforce a well−defined `look and feel` policy in UnixWare. An application developer, having defined frames (menus, forms, and text) in files, is free from having to program their display and user interactions. The shell−like language is processed by an interpreter and allows the developer to specify menu and form placement. FMLI allows application developers to customize specific applications easily and quickly without writing in C language code.

The best−known use of FMLI is the sysadm(1M) command, which is a form and menu based system administration interface.

Refer to the fmli(1) manual page and the chapter The Form and Menu Language in the book *Character User Interface Programming* for more information.

The menu Command

The UnixWare `menu` command is a menu and form interface generator that creates full−screen forms and menus for accepting user input and for displaying information. The form or menu to be displayed is specified in a form description file that contains the text the user will see as well as the descriptions of any input or selection fields provided on a menu. The output of the `menu` command is a file of shell variable assignment statements that can be used by a shell script.

Refer to the menu(1) manual page for more information about the menu command, and the menu(4) manual page for information about writing a form description file.

The ckitem Command

At a lower level, the UnixWare command `ckitem` may be used for interactions with a user. `ckitem` builds a menu and prompts the user to choose one item from a menu of items. It then verifies the response. Options for this command define, among other things, a prompt message whose response is a menu item, text for help and error messages, and a default value. Typically, this command is used in a shell script where look and feel is not a crucial concern, as it's not a full−screen interface.

Refer to the ckitem(1) manual page for more information.

Character User Interface APIs

Application programming interfaces are provided in UnixWare for developers who want to develop full−screen or menu− and form−based interfaces that operate on ASCII character terminals (including xterms under the X Window system) running on UNIX System V Release 4.2 and later, including UnixWare. Existing applications can be adapted to a character user interface front−end, and new applications can be designed from the start to take advantage of the screen management capabilities of curses and ETI.

The curses Library

The `curses` library routines give you a terminal−independent method of updating character screens with reasonable optimization. This library exists on most UNIX−based operating system variants, and so is the most portable character user interface API provided in UnixWare.

The `curses` library allows: overall screen, window and pad manipulation; output to windows and pads; reading terminal input; control over terminal and curses input and output options; environment query routines; color manipulation; use of soft label keys; terminfo access; and access to low−level curses routines.

See the curses(3curses) manual page, and the chapter Simple Input and Output in the book *Character User Interface Programming* for more information.

The ETI library

ETI is a set of screen management library subroutines (built on `curses`) that promote fast development of application programs that manipulate windows, panels, menus, and forms. ETI also includes functions to define help, error and other types of messages, and to display, and change messages quickly and easily. It is a C language toolkit used to build user interfaces for applications. ETI allows you to design a unique/customized user interface.

There are several chapters in the book Character User Interface Programming that can provide more information about programming using ETI.

Graphical User Interface Tools

UnixWare provides wksh to ease the writing of graphical user interface for software. The Windowing Korn Shell (wksh) offers a number of extensions to the standard Korn Shell [see ksh(1)] in order to create and manage graphical user–interface widgets. There are also a number of commands that allow the programmer to extend thewksh language by attaching C code dynamically, register C functions as new built–in commands, register new widget definitions, and call C functions directly from attached libraries. All normal ksh commands and features are also supported; wksh is a true superset of standard ksh.

See the manual page for wksh(1) and the book Windowing Korn Shell Programming for more information.

Graphical User Interface APIs

The UnixWare X Server is an adaptation of the server in the X Window System, version X11 Release 5. In addition to the X Window System, UnixWare provides the UnixWare Desktop, a desktop manager, as an easy–to–use and consistent graphical environment for users to perform typical daily tasks such as invoking applications. When using the UnixWare Desktop, what a user sees on the screen are windows and icons each representing an integral component of the system.

UnixWare provides programming tools and libraries for interfacing with the X Window System and with the UnixWare Desktop.

X Window System Libraries

In addition to the typical X Window System programming libraries, such as Xlib and the Athena widget set, UnixWare also provides the Motif toolkit. This enables application developers to write graphical programs to the Motif interface, which is popular with and familiar to many users.

See the book Motif Programmer's Guide for information on programming to the Motif interface. In addition, the book Motif Style Guide provides additional information on how to write effective graphical user interfaces using the Motif toolkit.

UnixWare Desktop Library

The Desktop application programming interface is toolkit–independent and can be used with MOTIF or other toolkits. The libDt library implements the UnixWare Desktop API. Programming to this library will enable you to write applications that are very tightly integrated with the UnixWare desktop, and take advantage of all of it's features. Routines are available to utilize the following facilities:

- folder manipulation
- file class management
- wastebasket
- on–line help

See the chapter The UnixWare Desktop Application Programming Interface in the book *Graphical User Interface Programming and Tools* for complete information on programming to this interface. Elsewhere in the bookGraphical User Interface Programming and Tools there are detailed discussions of the operation of the UnixWare Desktop and other levels at which you can integrate your applications with the UnixWare Desktop. A more detailed discussion of integration with the UnixWare Desktop may be found in UnixWare Desktop Integration.

Internationalization

This section provides a brief overview of the internationalization support provided in the UnixWare Software Developer's Kit. There are two types of support for internationalization provided in UnixWare: support for locales and support for wide and multibyte characters.

Locales

At any time, an executable program has a current `locale`, meaning a collection of information that describes the conventions appropriate to some nationality, culture, and language. Locales have names that are strings, and the only two standardized locale names are `"C"` and `""` (null). Each program begins in the `"C"` locale which causes all library functions to behave just like they have historically. The `""` locale is the implementation's best guess at the correct set of conventions appropriate to the program's invocation. (Of course, `"C"` and `""` can cause identical behavior.) Other locales may be provided by implementations.

For the purposes of practicality and expediency, locales are partitioned into a set of categories. A program can change the complete locale (all categories), or one or more categories leaving the other categories unchanged. Generally each category affects a set of functions disjoint from the functions affected by other categories, so temporarily changing one category can make sense.

For complete information, see the section Internationalization in the chapter *Complying with Standard C* in the book *Programming in Standard C and C++.*

Wide and Multibyte Characters

At first, the `internationalization` of ANSI C affected only library functions. However, the final stage of internationalization −−− multibyte characters and wide characters −−− also affected the language itself.

The basic difficulty in an Asian environment is the huge number of ideograms needed for I/O. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all of these encodings allow intermixing of regular single−byte characters with the ideogram byte sequences. Just how difficult it is to recognize distinct ideograms depends on the encoding scheme used.

The term `multibyte character` is defined by ANSI C to denote a byte sequence that encodes an ideogram, no matter what encoding scheme is employed. All multibyte characters are members of the so−called `extended character set.` (A regular single−byte character is just a special case of a multibyte character.) The only requirement placed on the encoding is that no multibyte character can use a null character as part of its encoding.

ANSI C specifies that program comments, string literals, character constants, and header names are all sequences of multibyte characters. See the section Multibyte Characters and Wide Characters in the chapter *Complying with Standard C* in the book *Programming in Standard C and C++* for more information about support for locales, including the setlocale(3C) function, and new and changed library routines.

Using Kernel Threads

UnixWare 2 is a symmetrical multiprocessing operating system. As such, it supports concurrent programming; this allows you to write a program so that the amount of time it takes to execute could be reduced by adding processors. The **threads** library is the programming interface provided to allow concurrent programming in UnixWare 2.

Concurrent Programming

With concurrent programming the programmer can specify sets of instructions that potentially can be executed in parallel and still provide correct results.

The advantages of this style of programming are:

- A powerful programming paradigm

  Programs are often written to emulate or respond to events in the real world. In the real world, concurrency is common and purely sequential events are the exception. Modeling such behavior is facilitated if the programming environment supports the notion of concurrency.

- Possible performance improvement

  If multiple processors are available, the program might be executed in less real time (than sequential execution) if more than one processor is working simultaneously. This is called true concurrency.

  Even on uniprocessor machines, there may be some performance gain from designing greater concurrency into the program. While one activity is blocked, others might still be executing.

  Thus, there is an advantage to concurrent programming even if the resources (processors) are not available to provide true concurrency and the application is only logically concurrent.

Concurrent programming has been available in the UNIX System since its inception via the process model. In the UNIX System, problems are solved not just by running programs but by running sets of programs; in some cases pre−existing tools or commands; or specifically written programs that work together (often concurrently) to solve the problem. Processes can communicate and synchronize with each other by mechanisms that include:

- pipes (named and unnamed)
- files and file/record locks
- signals
- messages
- shared memory (IPC style shared memory or mapped files)
- semaphores (IPC style)

What are Threads?

Starting with Release 2, UnixWare provides vastly expanded capabilities for concurrent programming via the threads library. These capabilities include:

- Facilities to define multiple threads of control to be run concurrently within a single process. Each thread is a set of instructions that is itself sequential but can be executed concurrently with other threads.
- A new, rich set of software mechanisms for coordinating and synchronizing the activities of the process' threads. These include:
  - mutual exclusion locks called **mutexes**; both recursive and not; both blocking and spinning.
  - reader−writer locks
  - counting semaphores (not the IPC semaphore system calls)
  - condition variables
  - barriers

- Features to control the level of concurrency and the scheduling of threads.
- Underlying operating system kernel support that enables the library to provide true concurrency (on multi−processor architectures), not just logical concurrency for threads.

Programming with Threads

For complete information on programming with the threads library, please refer to the chapter Programming with the Threads Library in the book *Programming with System Calls and Libraries*. This chapter contains in−depth descriptions of threads, concurrent processing, the programming environment for writing threaded applications, and how to debug your threaded applications.

Chapter 4
Developing Device Drivers for UnixWare

Introduction to Developing Device Drivers for UnixWare

There are a number of issues to be aware of when writing device drivers for UnixWare, or when porting device drivers from another UNIX operating system. This chapter gives a brief introduction to many of these issues and provides pointers to sources of information to help answer questions and provide examples of writing device drivers for UnixWare. If you are just starting out, it is a good idea to read the IHV Development Kit Roadmap first.

If you have not installed the IHV Development Kit package *ihvkit* from the UnixWare Software Developer's kit, it is a good idea to do so. This package contains the on−line book IHV Development Kit Roadmap as well as several tools to aid in writing, debugging, and packaging device drivers for UnixWare.

Device Driver Features

UnixWare provides a number of features for writing device drivers. These features include the ability to load and unload device driver modules on a running system, the ability to reconfigure device drivers without rebuilding the kernel, and a well−defined interface for writing your device drivers. These features are powerful resources for programmers, and help make it easy to write device drivers for and port device drivers to UnixWare. In addition, they can make your device drivers flexible and easy for your users to administer, and for your software packages to install.

Loadable Modules

The Dynamically Loadable Modules (DLM) feature allows you to add a device driver or other kernel module to a running system without rebooting the system or rebuilding the kernel.

The DLM feature

- reduces time spent on device driver development by streamlining the device driver installation process
- makes it easier for users to install device drivers from other vendors
- improves system availability by allowing device drivers to be configured into the kernel while the system is running
- conserves system resources by unloading infrequently used device drivers when they are not needed (when needed in the system, DLM loads the device drivers from disk)
- gives users the ability to load and unload device drivers on demand
- gives the kernel the ability to load and unload device drivers automatically
- requires device drivers that are going to be configured into the system as loadable modules to be converted to loadable form

See the chapter Loadable Modules in the book *Device Driver Programming* for a complete discussion of

Dynamically Loadable Modules with sample code.

Autoconfiguration

The autoconfiguration facility, introduced in UnixWare 2, allows you to configure your device drivers dynamically and automates the entire hardware configuration process to a far greater extent then ever before.

In the past, hardware configuration almost always required the manual setting of jumpers or DIP switches. More recently, vendors have started building software configurability into their boards and adapters, supplying custom, special−purpose software to perform the configuration.

The UnixWare autoconfiguration facility takes this process a step further by organizing and automating the hardware configuration process to a greater extent (as far as the given hardware will support). The problems that this feature addresses are:

- Inability of the system to deal automatically with autoconfigurable devices (such as EISA and PCI devices).
- No ability to change statically configured device parameters. When installation required parameters other than those configured, installation was impossible.

The UnixWare autoconfiguration facility was designed to make configuration as easy and automatic as possible, given the current capabilities of computer hardware and making allowances for backwards compatibility with older expansion boards.

When device parameter information is available from NVRAM (non−volatile RAM storage, as on MCA, PCI, and EISA systems), this information is used to configure the device drivers automatically. UnixWare provides facilities for adding, deleting, and modifying device parameters through an interactive tool called the Device Configuration Utility (DCU).

See the System Owner Handbook for more information on how to use the DCU. See the chapter Autoconfiguration in the book *Device Driver Programming* for a complete discussion of Dynamically Loadable Modules with sample code.

Device Driver Programming

When writing device drivers for UnixWare, or porting device drivers from another operating system to UnixWare, there are a number of issues that you should take into account. First, there are different types of device drivers that have different requirements. In addition, there are well−defined interfaces provided in the UnixWare system for writing your device drivers so that they interface cleanly with the kernel. Also, since the UnixWare kernel is multiprocessing, it is necessary to enable your device drivers for threads, which allow for concurrent processing.

Types of Device Drivers

PDI Device Drivers

The Portable Device Interface (PDI) is an architecture for device driver development that emphasizes the separation of hardware−dependent and hardware−independent pieces of device drivers. For information about writing device drivers to use the Portable Device Interface (PDI), see the programming guide, Portable Device Interface (PDI). This guide contains chapters that not only provide an overview of PDI and peripheral devices, but also explains what you need to know to write Host Bus Adapter (HBA) and target device drivers. Helpful hints can be found in the files in the directory in `/usr/src/ihvkit/pdi/hba.guidelines`. In addition, the routines specific to PDI are documented in the D2H, D3I, and D4I sections in the Device Driver Reference.

Video Device Drivers

For information about video drivers, also known as screen interface (SI) modules, see the Device Driver Programming guide. You can also refer to the on−line `README` files included in this kit. Once the kit has been installed, comprehensive SI information can be found the `README` file, which is located in `/usr/src/ihvkit/display`. Details about individual SI modules located in additional `README` files in the directories associated with each SI module.

Networking Device Drivers

Device drivers that operate Network Interface Cards (NICs) are another major type of device driver. They are a different sort of interface from video and storage device drivers, as they are typically for high−speed devices with no storage. In addition, UnixWare 2 makes use of Open Device Information (ODI) device drivers developed for NetWare for additional device driver support; thus, an ODI device driver can be **transmogrified** for use by the UnixWare kernel. For a detailed discussion of writing network device drivers, including information on transmogrifying ODI networking drivers, see the chapter Writing Network Drivers in the book *Device Driver Programming*.

Driver Kernel Interface (DDI/DKI)

The DDI/DKI is a mature interface between device drivers and the devices they control, and between device drivers and the rest of the kernel. For software conforming to the DDI/DKI, we make certain guarantees about portability of that software to future releases of UnixWare and UNIX System V Release 4. See the introduction in the DDI/DKI section.

Portable Device Interface (PDI)

The Portable Device Interface (PDI) is an architecture for block−oriented devices. The PDI presents a framework for device driver development that emphasizes the separation of hardware−dependent and hardware−independent pieces of device drivers. The routines described here complement those of the Device Driver Interface/Driver−Kernel Interface (DDI/DKI). While the DDI/DKI defines a standard interface between device drivers and the rest of the kernel, the PDI defines a set of routines for doing common tasks within drivers. See the chapter *Introduction to the Portable Device Interface* in the book Device Driver Reference as well as Device Driver Programming for more information on the PDI interface.

Data Link Provider Interface (DLPI)

The Data Link Provider Interface (DLPI) is an interface for device drivers that provide data link services, such as networking hardware device drivers. The DLPI provides a set of hardware−independent routines and defines message formats that the device driver must generate/process. The device driver writer supplies the hardware−specific portions of the device driver. These two pieces, in combination, create a functional DLPI device driver. See the chapter *Introduction to the Data Link Provider Interface* in the book Device Driver Reference as well as Device Driver Programming for more information on the DLPI interface.

Transport Provider Interface (TPI)

The Transport Provider Interface, or TPI, is a STREAMS−based service interface that maps to strategic levels of the Open Systems Interconnection (OSI) Reference Model. TPI supports the services of the Transport Layer for connection−mode and connectionless−mode services. One advantage to using TPI is its ability to hide implementation details of a particular service from the consumer of the service. This enables system programmers to develop software independent of the particular protocol that provides a specific service. See the chapter STREAMS−based Transport Provider Interface in the book *STREAMS Modules and Drivers* as well as Device Driver Programming for more information on the TPI interface.

Multithreading Drivers

The most accurate and up−to−date information about the routines used in multithreading device drivers for multiprocessor systems is in the Device Driver Reference and Device Driver Programming. These books discuss the use of spin locks, sleep locks, and event synchronization primitives for multithreading device drivers, and provide background knowledge in multithreading techniques.

For details about multithreading PDI device drivers, refer to Portable Device Interface (PDI), in the chapter on Host Bus Adapter (HBA) driver development. Information about STREAMS modules and device drivers in a multiprocessing environment is covered throughout the STREAMS Modules and Drivers guide.

Device Driver Installation

For device driver writers, installation means different things. If you are installing a device driver for a piece of hardware; for example, you'll have some hardware−related installation procedures to follow. When you install the device driver you've written on your computer for the first time, you probably will install it without the installation scripts recommended for customer use. When you do create the device driver package for customers, installation will take on a different meaning.

Using idtools

In UnixWare, device drivers (and other types of kernel modules) are packaged, installed, and configured into the system using a collection of configuration files, commands, and scripts known as the Installable Driver Tools, or idtools.

It is important to note that the idtools have automated much of what used to be manual editing of device driver configuration files. There are several benefits to automating this process, among them being decreased chances of total system failure because a single file has been lost or corrupted, fewer problems when installing a new device driver, and a much simpler process for removing installed device drivers.

Although you might create the configuration file without using idtools, once the file becomes part of a device driver, everything you do with the file from then on−−−from installing it, to rebuilding the UnixWare kernel, to removing the device driver from the system−−−should all be done using idtools.

Detailed information on each of the idtools commands can be found in the Section 1M manual pages in the Command Reference.

In addition, see the chapter Driver Installation and Tuning in the book *Device Driver Programming* for detailed information about and guidelines for using the idtools for building and installing device drivers, as well as reconfiguring and rebuilding your kernel.

Driver Software Packaging

Producing a commercial−quality device driver involves not only developing the device driver source code and testing it with the underlying hardware, but also includes creating related packaging and installation scripts. UnixWare offers a rich set of tools to help in packaging, installing, configuring, and tuning device drivers.

A Driver Software Package (DSP) can be divided into two distinct collections of files and scripts:

- The object module (`Driver.o`) and other system−specific configuration files including `Space.c`, `System`, `Master`, and `Node`.

- Packaging–related files including `prototype`, `pkginfo`, `postinstall`, `preremove`, and `request`, which assist in the installation and configuration process.

See the chapter Driver Packaging in the book *Device Driver Programming* for complete information about packaging your device driver for UnixWare.

Chapter 5
Integrating Software with UnixWare

Introduction to Integrating Software with UnixWare

When you develop software for a particular platform, or develop new software for that platform, it is a great help to your users to provide some level of integration with their operating system. While it is possible to distribute your software in such a way that the user has to use a command like `cpio` or `tar` to read it in, and then run some customized script to put the software in it's proper place, this approach largely ignores the facilities provided with a particular system for software management. Similarly, on a system with a graphical front end, it isn't advisable to distribute software that cannot be made available from that front end.

In UnixWare, there are a number of facilities provided for ease of use and administration. In the area of software management, there is a set of utilities that come with the system for packaging, installing, removing, and distributing software. In the area of user interfaces, there is a graphical Desktop that's distributed with UnixWare, which is the primary means by which end–users are expected to use the system. In addition, there are a number of features for security that allow users to perform specified administrative tasks without being root.

As you port software to, or write software for UnixWare, you're encouraged to take advantage of these features. Doing so will allow your software to be a better integrated part of the UnixWare system, and thus be better received and better understood by your users who are familiar with UnixWare, and who expect software to work consistently with the rest of the system.

Software Packaging

UnixWare comes with an advanced set of administrative facilities. Included is a set of software management tools that allow software to be packaged, installed, removed, distributed, and maintained in a consistent manner. While software can be copied into place by archiving utilities such as `tar` or `cpio`, there are a number of advantages to using the packaging tools provided with UnixWare and distributing your software in UnixWare's packaging format. Properly packaged software can be:

- installed via the UnixWare pkgadd(1M) utility
- installed via the UnixWare Desktop's `App_Installer` interface
- transferred to installation media via the pkgtrans(1) utility
- installed from an install server on your network via the pkginstall(1M) or the `App_Installer` interface
- distributed from an install server to another via the pkgcopy(1M) command
- sanity–checked using the pkgchk(1M) command
- removed from a system on which it's installed via the pkgrm(1M) command

Essentially, when you package your software correctly, it saves you from having to write additional code to administer the installation and uninstallation of your software. For a complete discussion of how to package your software, see the chapter Packaging Your Software Applications in the book *Software Development Tools*, which includes a number of examples. For information on distributing your software from a server on your network, see the section Setting Up and Administering an Install Server in the chapter *Installing Add−on Software* in the book *System Administration*.

UnixWare Desktop Integration

The UnixWare Desktop is a user−friendly graphical front end to the UnixWare operating system. It is built on top of the UnixWare X Server, which provides the graphic services. The Desktop provides an easy−to−use and consistent graphical environment for users to perform typical daily tasks such as invoking applications like spreadsheets and word processors. What you see on your screen are windows and icons each representing an integral component of your system. The Desktop provides several facilities that allow you to develop and provide graphical applications on your system. These facilities are the topics of this book.

On the Desktop, all resources are represented by folder windows and icons and can be manipulated using menus and a drag−and−drop mechanism. For example, open directories are represented as windows and files within those directories are shown as icons. Printing can be performed through menu selection or drag−and−drop.

The UnixWare Desktop provides several means to allow you to enhance the UnixWare Desktop by

- changing a file class or adding new ones
- changing the Desktop behavior of files in these file classes
- creating new icons for existing or new file classes

These mechanisms allow you to integrate your application into the UnixWare Desktop; with them, you create file types, associated icons, and default behaviors for those files. You can also change the behavior of the existing Desktop. This section introduces these capabilities.

The following sections provide an overview of what it means and what it takes to integrate your application with the UnixWare Desktop. They will introduce the necessary concepts for you to get started, and provide you with information sufficient to let you plan and design your software's integration with the Desktop. As with the other sections in this book, these sections do not provide a complete tutorial for performing this integration. However, the following sections do provide background and pointers to more complete sources of information that can be found in the documentation included with your UnixWare system.

The File Class Database

The UnixWare Desktop Manager partitions all files into a set of classes. Every file on the UnixWare system belongs to a class and each class has its own graphical representation (icon) and a set of properties. The file class database contains the file class descriptions and defines the icons that represent each file class.

The class database describes the properties of a desktop class. If you were to add one or more new icons on the UnixWare Desktop for your application, you would create a new class for each icon that fully describes the characteristics of the icon, including the icon file, the type of file that's associated with the icon, any menus you want to associate with the icon, what to do when the icon is selected, etc. See the section Defining New File Classes in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools* for complete information about updating the class database for your application.

The Icon Definition File

All icons provided with the UnixWare Desktop are defined by entries in the file class database. When the desktop manager opens a folder, it attempts to match each of the objects in the folder against entries in the database. If the desktop manager cannot match an object with a particular class, it defaults to a class associated with the file type.

Thus, in order to ensure that your application's icons are correctly identified and displayed, it's important to make sure that your icon definition file is correct.

When installing a package that is in Unix System V Release 4 pkgadd(1M) format, the application installer (`App_Installer` in the `Admin_Tools` folder) first looks for the presence of this icon definition file. Without this definition file, the installation of an application targeted for the Desktop may be successful, but the look and feel of the installation to the user will be different than that of the other applications available in UnixWare.

See the section The Icon Definition File in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools* for details about the format and location of this file.

Application Integration Check List

The following checklist is enumerates Graphical User Interface Programming and Tools, what should be provided along with your application in order to integrate it with the UnixWare Desktop. For more information refer to Graphical User Interface Programming and Tools.

- Icon file for your application when it is in a folder and when it is minimized. See the section Creating Icon Files in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools* and the section Creating the Icons for Drop Sites in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools.*
- Help files, if any, preferably in hypertext format. See the section Interface to On−line Help in the chapter *The UnixWare Desktop Application Programming Interface* in the book *Graphical User Interface Programming and Tools*.
- File class(es) to be used to class your application and files associated with it, if any. See the section Defining New File Classes in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools.*
- Icon definition file, which is used when your application is installed via `App_Installer` to install your application in a particular folder and the Help Desk, and to install the file class(es). See the section The Icon Definition File in the chapter *Integrating Applications with the UnixWare Desktop* in the book *Graphical User Interface Programming and Tools.*
- Message catalog for file classes and client messages, if any.

Sysadm vs. the Desktop

Older UNIX systems (for example, systems based on Unix System V Release 3 or Unix System V Release 4), relied heavily on the `sysadm` interface for performing many system management tasks. While this tool is still distributed with UnixWare, it has been de−emphasized. In UnixWare, the graphical Desktop is the preferred interface for system management tasks. If your software performs system management tasks, or if your software itself requires management, it is highly recommended that you integrate your software with the UnixWare Desktop.

If you require a character interface for a system administration task, it is still possible to use `sysadm`. See the chapter Customizing the sysadm Interface in the book *System Administration* for a complete discussion, with examples, of how to add tasks to the `sysadm` interface.

Security Issues

The UnixWare system employs the concept of *`permissions`* to control both the access that users have to information stored in files, and the ability of users to execute certain commands. Users who have been granted permission to execute administrative commands, or access system files, are said to be `privileged` users. A user can be granted all privileges, or only a subset. Permissions are particularly important in the area of system administration, because unauthorized changes to system files and use of commands can adversely affect the productivity of users and the usability of the operating system.

Owner Versus Root Permissions

For those users who are working through the UnixWare Desktop interface, UnixWare implements the concept of a system owner. When you install the operating system, a user is designated as the owner. Any user name (except those already reserved by the system, such as `sysadm` and `bin`) can be assigned as the system owner. The system owner is given the necessary permissions to do the day–to–day system administration tasks available through the UnixWare Desktop interface. These special permissions are described in the Desktop User Handbook and the System Owner Handbook.

However, once an owner leaves the UnixWare Desktop interface (by opening a Terminal window or logging in as a non–Desktop user) the traditional UNIX system permissions take effect. The special permissions of a system owner carry over to the shell interface, but they do not represent the full set of permissions that are available. The traditional UNIX system user who has the full set of permissions for doing administration is the `root` user.

When developing software for UnixWare, it's important to keep in mind these issues about security. If you're writing system administration software, it's important that you use privileges to enable your application. This allows the UnixWare Desktop owner, who may be a non–`root` user, to perform administrative tasks. The command used to grant privilege to a non–`root` user is tfadmin(1M).

Security Administration

System administrators and privileged users need to perform sensitive tasks, but because privileges are associated with processes, not user IDs (except for the special case of UID 0), it is not possible to directly grant privileges to users to perform these sensitive tasks. The Trusted Facility Management (TFM) tools provide the means to maintain a database of users and the commands they may execute with privilege.

TFM eliminates the need to place fixed privileges on a command (via `filepriv`) for it to execute with privilege for these users. An administrator who is in the TFM database must execute the `tfadmin` command to run these commands with the granted privileges. Users not in the TFM database cannot execute commands with privilege and will get an error if they attempt to use `tfadmin`.

TFM also gives you the ability to associate a set of privileged commands with a role. The privileged commands associated with each role are then associated with people assigned to that role. This saves you from having to define an identical set of privileged commands for each user. Instead, you define a role as containing the commands you desire, and then associate each desired user with the role. See the section TFM and Administrative Roles in the chapter *Trusted Facility Management* in the book *System Administration* for details on the definition and maintenance of roles.

The tfadmin Command

The `tfadmin` command allows administrators and privileged users to execute those commands with which they have been associated via the `adminuser(1M)` or `adminrole(1M)` commands

The system comes with a default TFM database that lists aliases for commands that require privileges, the privileges associated with them, and the roles that are capable of executing them. You will need to associate administrators with roles so these administrators will be able to execute these commands. A more complete discussion of these concepts can be found in the section The tfadmin Command in the chapter *Trusted Facility Management* in the book *System Administration*

The important concepts to remember about system security when developing software for UnixWare are:

- Administrative functions should be available from the UnixWare desktop.
- The UnixWare Desktop system owner is a trusted administrator of the UnixWare system.
- If you write administrative software, it should not depend on being run by `root`.
- You can assign privileges using the tfadmin(1M) command.
- A complete discussion of the concept of privileges can be found in the chapter Administering Privilege in the book *System Administration*
- A complete discussion of trusted facility management can be found in the chapter Trusted Facility Management in the book *System Administration.*