LynxOS User's Guide

LynxOS Release 4.0

DOC-0453-00

Product names mentioned in *LynxOS User's Guide* are trademarks of their respective manufacturers and are used here for identification purposes only.

Copyright ©1987-2002, LynuxWorks, Inc. All rights reserved.
U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

All rights reserved. No part of this *LynxOS User's Guide* may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photographic, magnetic, or otherwise, without the prior written permission of LynuxWorks, Inc.

LynuxWorks, Inc. makes no representations, express or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither LynuxWorks, Inc., nor its distributors, nor its dealers shall be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights which vary from state to state within the United States of America.)

Contents

Preface

This *LynxOS User's Guide* provides information about basic system administration and kernel configuration for the LynxOS real-time operating system from LynuxWorks, Inc. This guide covers a wide range of topics, including tuning system performance, and creating kernel images for embedded applications.

This document assumes that its audience has a basic understanding of the UNIX environment.

While this document provides information for a variety of readers--system administrators, network administrators, developers, and end-users of LynxOS--many of the tasks described in it require root privilege or access to other information typically given only to system administrators.

For More Information

For information on the features of LynxOS, refer to the following printed and online documentation.

- *Release Notes*

  This printed document contains details on the features and late-breaking information about the current release.

- *LynxOS Installation Guide*

  This manual details the installation instructions and configurations of LynxOS and the X Windows System.

- *LynxOS Networking Guide*

  This guide contains configuration and usage information on the networking capabilities in LynxOS. It provides information on supported protocols such as TCP/IP, NFS, DHCP, etc.

- *Writing Device Drivers*

  This guide contains details on writing device drivers for LynxOS.

- Online information

  The complete LynxOS documentation set is available on the Documentation CD-ROM. Books are provided in both HTML and PDF formats.

  Updates to these documents are available online at the LynuxWorks web site: `http://www.lynuxworks.com`.

  Additional information about commands and utilities is provided online with the **`man`** command. For example, to find information about the GNU gcc compiler, use the following syntax:

  ```
  man gcc
  ```

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to file names and commands are case sensitive and should be typed accurately.

Type of Text	Examples
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *LynxOS User's Guide*.

Environment variables, file names, functions, methods, options, parameter names, path names, commands, and computer data
Commands that need to be highlighted within body text, or commands that must be typed as is by the user are **`bolded`**.
	`ls`
	`-l`
	`myprog.c`
	`/dev/null`
	`login: myname`
	`# cd /usr/home`

Text that represents a variable, such as a file name or a value that must be entered by the user	`cat filename` `mv file1 file2`

Type of Text	Examples

Blocks of text that appear on the display screen after entering instructions or commands	`Loading file /tftpboot/shell.kdi into 0x4000`
	`.....`
	`File loaded. Size is 1314816`
	`Copyright 2000 LynuxWorks, Inc.`
	`All rights reserved.`
	
	`LynxOS (ppc) created Mon Jul 17 17:50:22 GMT 2000`
	`user name:`

Keyboard options, button names, and menu sequences	**Enter**, **Ctrl-C**

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.

CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products, and LynuxWorks news groups.

LynuxWorks U.S. Headquarters

Internet: support@lnxw.com
Phone: (408) 979-3940
Fax: (408) 979-3945

LynuxWorks Europe

Internet: tech_europe@lnxw.com
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

World Wide Web

http://www.lynuxworks.com

CHAPTER 1 *Introduction*

About LynxOS

LynxOS is a UNIX-compatible, POSIX-compliant, real-time operating system. It is designed to be used in time-critical applications where predictable real-time response is crucial. Its full range of functionality facilitates the development of custom embedded (function-specific) systems in both native and cross development environments.

LynxOS Features

- Multiprocess and multithreaded environment
- Sophisticated memory management through hardware Memory Management Unit (MMU)
- Configurable demand-paged virtual memory
- Hierarchical, UNIX-like file system
- Modular scalable architecture
- Kernel threads
- Network File System (NFS)
- Industry standard Networking (TCP/IP)
- Remote Procedure Calls (RPC)
- Support for diskless clients
- X11R6 and Motif graphical user interface
- ELF applications and shared libraries

- Industry standard GNU tools, UNIX-like utilities and UNIX-like shell scripts
- ROM-able kernel

LynxOS Documentation

Please refer to “For More Information” in the Preface for a list of primary LynxOS documentation. In addition to online man pages, LynxOS also comes with a suite of reference guides.

Additional Documentation Resources

LynuxWorks recommends any of a number of commercially available references for more information about advanced real-time programming, networking, industry standard tools, and the software engineering processes. The lists of documents that follow offer the user a starting point in these areas; however, these references should not be viewed as a complete list of source material.

General UNIX Titles

- Frisch, Aeleen. 1996. *Essential System Administration*. O’Reilly & Associates.
- Hunt, Craig. 1998. *TCP/IP Network Administration*.O’Reilly & Associates.
- Peek, Jerry, et al. 1997. *Learning the Unix Operating System (Nutshell Handbook)*. O’Reilly & Associates.

Linux Titles

- Welsh, Matt. 1999. *Running Linux*. O’Reilly & Associates.

Programming Titles

- Harbison, Samuel, and Guy Steele. 1994. *C: A Reference Manual*. Prentice Hall.
- Kernighan, Brian, et al. 1998. *The C Programming Language*. Prentice Hall.
- Libes, Don. 1994. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O’Reilly & Associates.

- Loukides, Michael Kosta. 1997. *Programming with Gnu Software*. O'Reilly & Associates.
- Oram, Andrew, et al. 1991. *Managing Projects with Make*. O'Reilly & Associates.
- Ouillane, Steve. 1995. *Practical C++ Programming*. O'Reilly & Associates.
- Ouillane, Steve. 1997. *Practical C Programming*. O'Reilly & Associates.
- Stevens, W. Richard. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley.

POSIX Titles

- Butenhof, David. 1997. *Programming with POSIX Threads*. Addison-Wesley.
- Gallmeister, Bill. 1995. *POSIX 4: Programming for the Real World*. O'Reilly & Associates.
- Lewine, Donald. 1991. *POSIX Programmer's Guide*. O'Reilly & Associates.
- Nichols, Bradford, and Dick Buttlar. 1996. *Pthreads Programming*. O'Reilly & Associates.

General Software Titles

- Brooks, Frederick. 1975/1995. *The Mythical Man-Month, Essays on Software Engineering*. Addison-Wesley.
- Humphrey, Watts S. 1989. *Managing the Software Process*. Addison-Wesley

About POSIX

POSIX, the **P**ortable **O**perating **S**ystem **I**nterface for UNI**X**, is both an IEEE Standard (IEEE 1003.1) and an ISO Standard (ISO 9945-1). Although based on UNIX, the standard can be adapted to other operating systems. POSIX is a high-level programmer interface definition of portable operating systems services.

The primary benefit of developing POSIX-conformant software is that it promotes platform (hardware) independent code, aiding in source code portability.

LynxOS and POSIX Standards

LynxOS conforms to the following POSIX standards:

- POSIX 1003.1 - the operating system interface standard to ensure application portability
- POSIX 1003.1b - the base real-time extensions to POSIX 1003.1
- POSIX 1003.1c - the threads extensions to POSIX 1003.1

Benefits of POSIX

LynuxWorks views POSIX as a significant benefit to customers allowing application developers and system integrators the ability to:

- Get products to market faster
- Lower the cost of getting products to market
- Preserve software development investment
- Recycle software
- Gain a higher level of software verification
- Integrate software more easily
- Extend POSIX-conformant software with less error

CHAPTER 2 *Getting Started*

This chapter provides a quick overview of basic LynxOS usage and concepts, including:

- Starting and shutting down the LynxOS Development Environment
- Starting and exiting from the X Windows
- Using PosixWorks Desk
- Using basic UNIX commands
- Creating file systems on floppy disks
- Making backups
- Starting the LynxOS environment for cross development systems

LynxOS Packages

The base LynxOS *Windows cross development* package includes the following CD-ROMs:

- Open Development Environment for Windows (ODEW)

  This CD-ROM contains:

  - Cross Development Kit for Windows (CDK)
  - Open Development Environment (ODE)
  - Board Support Package (BSP)
- Messenger (only included if required by a BSP)
- Additional Components CD-ROM containing Linux ABI Compatibility Layer, GNU Zebra Routing Package, OpenSSL Encryption Package

The base LynxOS *UNIX-hosted cross development* package includes the following CD-ROMs:

- Cross Development Kit (CDK)
- Open Development Environment (ODE)

  This CD-ROM also contains the Board Support Package (BSP) for a particular target.
- Additional Components containing Linux ABI Compatibility Layer, GNU Zebra Routing Package, OpenSSL Encryption Package
- Messenger (only included if required by a BSP)

The *native development environment* includes the following CD-ROMs:

- Open Development Environment (ODE)

  This CD-ROM also contains the Board Support Package (BSP) for a particular target.
- X & Motif
- Messenger (only included if required by a BSP)
- Additional Components CD-ROM containing Linux ABI Compatibility Layer, GNU Zebra Routing Package, OpenSSL Encryption Package

Starting and Stopping LynxOS

When powered on, systems running LynxOS boots to a command line prompt. If password access is enabled, users are prompted to enter a login name and password. Depending on the system's user access and privileges configuration, the initial working directory is either `root` or the user's home directory. The **`pwd`** command displays the current working directory.

On a cross development system where LynxOS is installed on top of an existing operating system (Windows, Linux, Solaris, etc.), the user must navigate to the LynxOS installation directory and set the LynxOS environment with a SETUP script. The script `SETUP.bash` is used for bash shells, and `SETUP.csh` is used for C shells. Refer to "Setting the Cross Development Environment" on page 12 for details.

To prevent system corruption, LynxOS must be powered-off with a predefined shutdown sequence. To properly shutdown LynxOS, use the following command:

```
# reboot -h
```

The **-h** option (halt) means that the system is not ready to power off until the following message indicates disk activity has stopped:

```
**** LynxOS is down ****
```

At this point, it is safe to power down the system.

Starting and Stopping X Windows

The native development configuration of LynxOS includes X11R6 and Motif (referred to as the "X Window System" or just "X Windows"). The X and Motif Package for LynxOS cross development systems is available separately. X Windows provides an industry standard graphical user interface (GUI) based on X11R6. The following command starts X Windows:

```
# startx
```

The keyboard sequence **Ctrl-Alt-Backspace** exits X Windows.

Using PosixWorks Desk

PosixWorks Desk provides quick access to LynxOS tools and utilities. Tool bars, buttons and icons facilitate management of the development environment. The folder-style file manager allows users to drag and drop files. Users can optionally install PosixWorks Desk when they install X Windows. Pressing the **Help** button on the main tool bar provides detailed information on PosixWorks Desk supported features.

Basic LynxOS Commands

Most basic LynxOS commands are identical to UNIX commands. This section briefly describes these basic LynxOS commands.

Further information about these commands is also available through LynxOS man pages. LynxOS man pages are accessed using the following command syntax: **man** *command*, where *command* is any LynxOS command verbatim. For more information, see "LynxOS Man Pages" on page 9.

Users should familiarize themselves with the following basic LynxOS commands:

Table 2-1: Basic LynxOS Commands

Command	Description
`cd` *`directory`*	Changes the current working directory to another specified *`directory`*.
`cd ..`	Changes the current working directory back up one level (`..`).
`cp` *`file1 file2`*	Copies the contents of *`file1`* to *`file2`*, creating a new file named *`file2`*, while preserving the contents and name of *`file1`* as well.
`find . -name` *`file`* `-print`	Finds files in the current working directory and its subdirectories with names containing expressions matching `-name` *`file`*, and prints these file names to the screen.
`gcc` *`file`*	Invokes the GNU C compiler (**`gcc`**), and compiles the executable code in the specified *`file`*.
`less` *`file`*	Displays a specified *`file`*'s contents, one screen at a time. • Pressing the **Enter** key scrolls the screen down to the next line. • Pressing the keyboard **Space Bar** scrolls the screen down to the next page. • Pressing the **`y`** key scrolls the screen up to the previous line. • Pressing **ctrl-B** on the keyboard scrolls the screen up to the previous page. Also, the **`less`** option has a help facility that describes additional options, such as string searches, etc.
`ls`	Displays the current working directory's contents and file information.
`man` *`subject`*	Displays the man page(s) for the specified *`subject`* (command, utility or tool name).
`mkdir` *`directory`*	Makes a directory with the specified *`directory`* name.
`mv` *`file1 file2`*	Moves contents of, and renames, *`file1`* to *`file2`* (*`file1`* is deleted).
`pwd`	Displays the path of the current working directory.
`rm` *`file`*	Removes the specified *`file`*.
`rmdir` *`directory_name`*	Removes /deletes specified directory.

Table 2-1: Basic LynxOS Commands (Continued)

Command	Description
`vi file`	Invokes the **`vi`** (visual) text editor, and creates a file with the specified *`file`* name.
`reboot -a`	Reboots LynxOS; the **`-a`** argument is optional, and is used to reboot the system in *multi-user* mode.

LynxOS Man Pages

LynxOS provides man (manual) pages for information and help with commands, utilities and tools. All LynxOS man pages are called with the syntax:

```
man subject
```

Where *`subject`* is any LynxOS command, utility or tool name. Users can enter the `man` command from anywhere on their systems.

LynxOS man pages are text files located in the `/usr/man` directory.

The LynxOS `man` command uses the `less` command to display the information one page at a time (see the description of `less` in the previous section). LynxOS man pages have the same user interface functionality as read-only files - users can view a text file and perform some operations on the file, such as scrolling and string searches, but cannot edit them.

While viewing the man page with the `man` command, users can get help with available viewing options by pressing the **h** key. Pressing the **q** key then **Enter** exits the help session and returns the user to the man page.

If a user is interested in a topic but is not sure what the exact commands are, a `man` search can be done for a keyword using the `-k` option:

```
# man -k keyword
```

The `man` command with a `-k` option is the least restrictive type of search - it is not case-sensitive, the keyword can be within the description of one or many commands, options, or the command name itself (as shown in the example command and its screen return below):

```
# man -k less
less (1) - interactive paginator files, viewing
lesskey (1) - specify key bindings for less
ltgt (3) - test for floating-point less-than or greater-
than
```

```
slaveboot (1) - transfers a kernel image to a diskless
SCMP client
```

Users can also use the more restrictive `-f` option with the `man` command --it is case sensitive--as shown in the example command and its screen return below:

```
# man -f less
less (1) - interactive paginator files, viewing
```

If neither of these options helps a user find information, the following command displays an alphabetical list of all LynxOS man pages:

```
# less /usr/man/windex
```

Creating File Systems and Making Backups

Users can create file systems and make backups to archive important data. The following steps detail creating a file system on a high-density floppy device, copying files to the floppy device, and then unmounting the device:

1. Insert an empty, high-density floppy disk into the floppy disk drive.
2. Format the floppy disk in the `fdd1440.0` disk drive with the `fmtflop` command and `-v` argument (verbose mode: displays messages as the floppy is being formatted):

   ```
   # fmtflop -v /dev/fd1440.0
   ```

3. Create a LynxOS file system on the floppy disk with the `mkfs` command:

   ```
   # mkfs -v /dev/fd1440.0
   ```

4. Mount the `fd1440.0` floppy disk's file system with the `mount` command onto the mount point directory, `/mnt`, by entering the following command:

   ```
   # mount /dev/fd1440.0 /mnt
   ```

NOTE: There can only be one file system mounted on a mount point (`/mnt` in this example) at any given time. Use the `umount` command to unmount a mounted file system.

---

5. Copy any number of files to the mounted floppy disk file system using the following command syntax, where *`directory`* represents any

directory, and *file.abc* and *file.xyz* represent any number of files from that directory:

```
# cp /directory/file.abc file.xyz /mnt
```

6. With the **ls** command, validate that the contents were copied to the disk.

```
# ls /mnt
```

7. Unmount the floppy disk drive when finished:

```
# umount /mnt
```

Making Backups

The **tar** archive utility is used to make backups of important data files on a variety of media. The type (s) of backup media chosen depends on both the development platform, and the amount of data that is to be backed up.

The following commands are examples of backing up data for a number of common configurations.

In the example below, the user's home directory is named `/usr/home/mystuff`. To back up the home directory's contents to a file named `backup.tar`, the following syntax is used:

```
# tar -cvf backup.tar /usr/home/mystuff
```

The `-cvf` arguments direct `tar` to perform the following actions:

- **c** - create a new tar archive
- **v** - provide verbose screen return (optional)
- **f** - direct the output to a file located in the current directory.

To back up directly to a device instead of a file, direct the output to a device node in the `/dev` directory. For example,

```
# tar -cvf /dev/device /usr/home/mystuff
```

To list the archival contents of the `backup.tar` file, the following syntax is used:

```
# tar -tvf backup.tar
```

In this example, `-t` is the argument that directs `tar` to list the archival contents of the `backup.tar` file.

To extract archival data in the file `backup.tar` and place it into `/usr/home/mystuff`, the following syntax is used:

```
# tar -xvf backup.tar /usr/home/mystuff
```

In the previous example, `-x` is the argument that directs `tar` to extract the archival contents of device `/dev/device`.

NOTE: If the target directory (in the example above, `/usr/home/mystuff`) already exists in the user's home directory, the `tar` command with the `-x` argument overwrites it with the archival data file (in the example above, `backup.tar`).

LynxOS Cross Development Environment

The term *Cross development* is the process of developing an application or kernel on a host system configuration that is different from the configuration of the target system (where the application is to be deployed).

The cross development environment provided in LynxOS includes compilers, linkers, libraries, and other development tools specific to LynxOS. The LynxOS cross development environment allows the developer the flexibility of creating LynxOS applications and kernels from a variety of platforms.

When working on a LynxOS cross development system, users must set their host system environment to use the LynxOS cross development tools before running any Makefiles or compiling applications. The cross development environment, once set, provides users the complete functionality of a native development system. The advantages of this approach is that there is no need to establish a dedicated machine for cross compiling and cross debugging an application. The application can be recompiled easily for different platforms, i.e., x86 and UNIX.

Setting the Cross Development Environment

Refer to the *LynxOS Installation Guide* for information on installing the Cross Development Kit (CDK) CD-ROM.

Before developing LynxOS applications on a cross development host, users must set the LynxOS cross development environment with a provided setup script. Two setup scripts are provided: **`SETUP.bash`**, which sets the `bash` shell environment; and **`SETUP.csh`**, which sets the C shell environment. Users must run one of these setup scripts, depending on their preferred shell, before compiling or linking LynxOS applications.

These setup scripts allow users to develop applications and run Makefiles as if they were on a native development system. With the exception of running the setup script, there is no operational difference between the application development process on a cross development or native development system.

The `SETUP.bash` and `SETUP.csh` scripts add the environment variable **`ENV_PREFIX`** to a user's **`PATH`**. The `ENV_PREFIX` variable specifies the directory on the host where the LynxOS development tools are located. This allows users to create their applications with LynxOS-specific compilers, linkers, and libraries. `ENV_PREFIX` is set to the first entry of a user's `PATH` to ensure that users create applications with LynxOS-specific tools.

Setting up Cross Development on UNIX Hosts

To develop LynxOS applications on a UNIX cross development host, run the setup script as follows:

1. Change to the LynxOS cross development directory. This is **`/usr/lynx/`***`release/platform`*, where *`release`* is the LynxOS release number, and *`platform is`* the target architecture. For example:

   ```
   # cd /usr/lynx/4.0.0/x86
   ```

2. Run the required setup script. For example:

   For C shells:

   ```
   # source SETUP.csh
   ```

   For `bash` shells:

   ```
   # . SETUP.bash
   ```

3. The setup script directs the user to the detected LynxOS development tools path. If this path is correct, type **`y`**. Otherwise, type **`n`** and enter the correct path to the LynxOS cross development tools directory.

   Once the setup script has run, users can begin to develop LynxOS applications on the cross development host.

Setting up Cross Development on Windows Hosts

1. From a DOS prompt, open a bash shell. For example, type:

   ```
   c:\ bash
   ```

2. Change to the LynxOS cross development directory. This is **/usr/lynx/***release/platform*, where *release* is the LynxOS release number, and *platform* is the target architecture. For example:

   ```
   # cd /usr/lynx/4.0.0/x86
   ```

3. Run the required setup script. For example:

   - For C shells:

     ```
     # source SETUP.csh
     ```

   - For `bash` shells:

     ```
     # . SETUP.bash
     ```

4. The setup script directs the user to the detected LynxOS development tools path. If this path is correct, type **y**. Otherwise, type **n** and enter the correct path to the LynxOS cross development tools directory.

Once the setup script has run, users can begin to develop LynxOS applications on the cross development host.

The Application Development Process

The figure below shows the basic steps in the application development process.

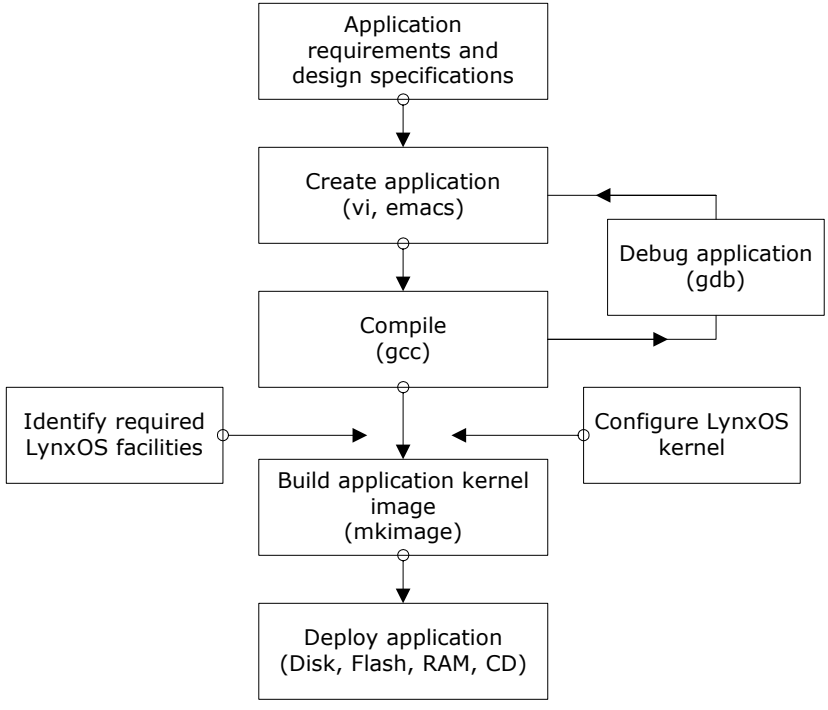

Figure 2-1: Application Development Process Overview

Creating Source Code with vi Text Editor

Users familiar with UNIX operating systems should already be familiar with the **vi** text editor. LynxOS provides vi as a part of its standard distribution.

NOTE: The text editor **`emacs`** is also supported by LynxOS for application development in native development environments.

For users unfamiliar with vi, this section provides a quick list of important functions.

`vi` is a full screen text editor used to create and modify ASCII files. There are two main modes in vi - *command mode* and *input mode*. Command mode is used for issuing short cursor placement or text selection commands. Input mode is used to add, change, or delete text in a file.

The following example shows how to create a text file, in this case a simple C program, using vi (other C program source files can be found in `/src/examples`):

1. Create and open a file by typing the following line:

   ```
   $ vi HelloWorld.c
   ```

2. **`vi`** opens the new file `HelloWorld.c` in command mode. Input mode is set by pressing **`i`**.
3. Type some characters.
4. Return to command mode by pressing the **`Esc`** key.
5. Close vi, and save all changes by entering the following:

   ```
   :wq
   ```

NOTE: To quit a file, discarding any changes, use the following command after pressing **`Esc`**:

```
:q!
```

To quit vi after opening the file as read-only (making no changes), enter the following command:

```
:q
```

Basic vi Commands

The following table details basic vi navigation and editing commands used in *Command Mode*:

Table 2-2: Basic vi Commands (Command Mode)

To move the cursor	
h	Move cursor left one character
l	Move cursor right one character
k	Move cursor up one line
j	Move cursor down one line
To delete or undo	
x	Delete character at cursor
dd	Delete row at cursor
u	Undo previous action
To switch from command mode to input mode	
i	Enters input mode and starts insertion *before* the character at the cursor.
a	Enters input mode and starts insertion *after* the character at the cursor.

Example vi C Program

Begin vi as shown previously and enter the following text:

```
#include <stdio.h>
main ()
{
int index = 10;
printf("\n\n****************\n");
printf("Hello World! \n");
printf("****************\n\n");
}
```

Now go into command mode and save the file. This example file is used in the following section, “Compiling and Linking Source Code” on page 18.

Compiling and Linking Source Code

After entering and saving the source code file, the user can create an executable application by compiling and linking it with the LynxOS GNU C compiler. To compile the `HelloWorld.c` file, enter the following command:

```
$ gcc HelloWorld.c
```

If the program compiled without errors a default executable file called **`a.out`** is generated.

Users can rename the executable to something other than `a.out` by using the **`-o`** *`name`* option as shown in the example below:

```
$ gcc -o HelloWorld HelloWorld.c
```

The resulting executable is the file `HelloWorld`. To execute the file, enter the following command:

```
$ ./HelloWorld
```

The executable prints the following text:

```
****************
Hello World!
******************
```

Debugging Source Code

LynuxWorks has two debuggers available for use with LynxOS:

- The GNU Debugger (GDB) - Provided as a standard part of the LynxOS development environment. For additional information, see the *Total/db User's Guide*.
- TotalView - Offered as an optional separate product; for more information on TotalView, see the *TotalView User's Guide* and *TotalView Supplement for LynxOS Users*.

Since TotalView is optional, only **`gdb`** is detailed below.

First, for GDB to function properly, the source code being debugged must be compiled using the **`-g`** GDB option as part of the compile command. This option adds debugging information to the executable. For example, to compile a source code file named `HelloWorld` for debugging with GDB, enter:

```
$ gcc -g -o HelloWorld HelloWorld.c
```

To start GDB, enter the following command:

```
$ gdb ./HelloWorld
```

To quit GDB, enter the following command:

NOTE: The GDB prompt is `(gdb)`.

```
(gdb) quit
```

GDB does not initialize or set any default breakpoints. A breakpoint makes a program stop whenever a certain point in the program is reached during execution. Users must provide the initial breakpoint. However, it is usually convenient to set a breakpoint at **`main`**, which is done using the following command:

```
(gdb) break main.
```

GDB includes a command recognition feature that requires users to enter only as many characters of any command as make the command unique. For example, if a user wants to enter the **`help`** command, it is only necessary to type **`he`**, then press the **Enter** key for GDB to bring up the help list of GDB commands and their options.

Some basic GDB commands are listed below:

Table 2-3: Basic GDB Commands

Command	Description
`help`	Lists GDB commands and their options. These command and option names can also be used as arguments to the `help` command for a display of detailed information about themselves.
`list`	Displays the source code.
`step`	Steps to the next line of source code.
`break` *`option`*	Sets a breakpoint at a specified line number or function.
`clear` *`option`*	Clears all breakpoints or the specified breakpoint *`option`*.
`inspect` *`expression`*	Shows the value of a data structure.
`run`	Runs program to completion or to the next breakpoint.
`quit`	Quits GDB.

Creating Custom Kernels

Users can customize individual LynxOS kernels specific to the requirements of their application. The LynxOS kernel can be optimized for maximum memory footprint economy, an advantage when memory constraints or costs are a key product requirement.

Unnecessary kernel components and/or features can be removed based upon specific application requirements.

For more information on LynxOS kernel customizations, see Chapter 7, “Customizing the Default LynxOS Kernel Configuration” on page 85.

Making Final Images

The LynxOS **`mkimage`** utility can be used to create a final application or LynxOS kernel image, and LynxOS facilities for run-time. The `mkimage` man page and Chapter 8, “Creating Kernel Downloadable Images (KDIs)” describe this process in detail.

Identifying LynxOS Facilities for Run-Time

Users can choose to distribute their final software product as a ROM- or RAM-based file systems containing a LynxOS kernel and an application. Users can also choose to distribute their software product as a LynxOS kernel, an application, or some subset of a development facility packages on more conventional storage media such as a disk or CD-ROM.

To distribute LynxOS in any of these configurations, users must purchase appropriate run-time royalty licenses from LynuxWorks.

NOTE: The number of run-time images that can be created depends on the specific license agreement with LynuxWorks. Users should refer to their specific customer agreements for details, feature capabilities, allowed time frame, and reporting procedure.

CHAPTER 3 *LynxOS System Administration*

System Administration Tasks

When a LynxOS system is used heavily by several users either for developing software and/or running applications, it is necessary to perform the following system administration tasks to maintain performance level:

- Adding and removing system devices and terminals
- User support, including security issues
- Spooling system configuration
- System shutdown and reboot
- Disk space maintenance and backup

Using the setup Utility

The **`setup`** utility is used through the initial configuration and any subsequent reconfiguration. To run `setup`, log in to the LynxOS system as **`setup`**, then type **`setup`** at the command prompt. The `setup` utility allows users to perform the following tasks:

- Add a password for the **`setup`** and **`root`** users
- Set up symbolic links for floppy drives (x86 and certain PowerPCs)
- Enable multiple consoles (x86 and certain PowerPCs)
- Enable **`ttys`** for login
- Add/delete users to/from the system
- Enable virtual memory
- Set time, time zone and daylight savings support

- Enable Write-Through Cache

The `setup` utility is a self-documented, menu-driven program. Read all the instructions displayed by `setup` before making selections.

Managing User Privileges

Many of the programs described in this chapter are available only to users with root access. Root users are often called the *superuser* because they have more system privileges than other users. The root user has the user ID number `0`.

After completing initial system setup, LynuxWorks recommends that a root password be created. Securing the root account protects the system from unwanted intrusion or use from unauthorized users.

Understanding the /etc Directory Contents

The files in the `/etc` directory are routinely used for system maintenance. The `/etc` files that need to be set up for each instance of LynxOS to run properly according to its configuration are described below.

Table 3-1: /etc Directory Contents

File	Description
`fstab`	Contains a list of devices with file systems usually mounted while the system is running. Each line contains five colon-separated fields (listed below) that display information about devices and their mountpoints: *`device : mountdir : type : freq : passno`* The `fstab` file needs to be accessed by `/bin/rc` during mounting and file system checks. See the `fstab` file format man pages; see man pages for the `mount` and `umount` utilities for more information.
`group`	Contains information listed in the four colon-separated fields below, where *`group_name`* is the user group's name; `*` is a vestigial artifact of previous operating systems; *`GID`* is the user group's identification number; and *`users_names`* is the list of groups that have access to this group's files: *`group_name : * : GID : users_names`* This file is used to add new user groups by inserting a line with the syntax shown above.
`motd`	Contains the messages of the day from the system administrator. These messages are automatically displayed upon user login.

Table 3-1: /etc Directory Contents (Continued)

File	Description
`mtab`	Contains information about currently mounted file systems. `mtab` is not readable or editable by administrators; it is a system file maintained by the `mount` and `umount` utilities. The `mtab` file, if it does not already exist, is created by `mount` during the initial file system mounting. The file contains a record of mounted devices, but its existence is not critical for proper system operation. See the `mtab` man page for more information.
`nodetab`	Contains a list of standard nodes automatically created with the **`mknod -a`** command. Each line displays the information about nodes in the colon-separated fields listed below: *`name : type : major : minor : perm`* For more information, see the `nodetab` and the `mknod` man pages.
`passwd`	Contains information describing each user known to the system. Each user is identified by an individual user ID that the `passwd` file maps to the user name. Each line displays information in the seven colon-separated fields listed below: *`name : passwd : uid : gid : info : dir : shell`* For more information, see the `passwd` man page in Section 5 of the man pages. The `passwd` file must exist. It is used by several system programs to relate items of information about a user to each other. It is also consulted by the `login` program to verify a user name.
`printcap`	Contains information about the various printers available on the system. Users can set up either local or remote printers. The **`lpr`** and **`lpd`** commands use this file whenever a print job is queued. For more information on this file, see the `printcap` man page.
`starttab`	Contains information read by `init` at system boot time. The `starttab` file gives the default `umask` value; the program to use for the single-user shell at boot time; the name of the system initialization file (usually `/bin/rc`); and the default data, stack, and core limits for all processes. For more information, see the `starttab` man page. The `init` process is the only system process that reads the `starttab` file. If the `starttab` file does not exist, `init` uses its default values.

Table 3-1: /etc Directory Contents (Continued)

File	Description
`tconfig`	Contains descriptions of serial port configurations; It serves the same purpose as `/etc/gettytab` on BSD-based systems. The `tconfig` file is referenced by **`stty`**, **`tset`**, and other programs to prepare a terminal for some special use. Each entry consists of one or more lines that assign values to various parameters. For more information, see the `tconfig` man page. The `tconfig` file is very important. Various utility programs (`tset` in particular) do not operate if the file cannot be found.
`termcap`	Contains descriptions of software features for various specific terminal devices. Each entry consists of one or more lines that assign values to various parameters. Terminal descriptions are read by the vi text editor, the shell, and other programs that must manipulate the terminal using software command sequences. The `termcap` file should exist, and should contain descriptions of all terminals likely to be used with the system.
`ttys`	Contains a list of serial ports to be recognized by `init` as valid for login. Each line displays information about its respective serial port or ATC virtual terminal in the five colon-separated fields listed below: *`device : flag : config : terminal : login`* For more information on the `ttys` file, see the `ttys` man page. The `ttys` file must exist for `init` to control multi-user operations.
`utmp`	Contains entries describing active login sessions. The file consists of records (not accessible for reading) containing the user name, host of origin, and login time for each active session. Each record corresponds to an entry in `ttys`. The `utmp` file must exist for the **`who`** utility to function. The file is created, if possible, by the `login` program if it has been removed.

Creating Device Nodes

Under LynxOS, all peripheral devices are accessed through device nodes. Each device *must* have an associated device node file.

The standard LynxOS image comes with all required device nodes pre-installed on the file system, ready to use. However, if new devices are added to a system, or if a new file system is created on another disk or diskette as the root file system, it becomes necessary to create new device nodes. The **`mknod`** program creates device

nodes for any character or block device, or named pipe (FIFO). The syntax for making a single character or block device node in the `nodetab` file is as follows:

```
# mknod name mode major minor
```

The italicized items are replaced by the following values:

- *`name`* is a LynxOS path name for the new device node. Device nodes can be created in any directory, but they are usually placed in **`/dev`**.
- *`mode`* represents the transfer interface mode with either the letter **`c`** for a character device, or the letter **`b`** for a block device.
- *`major`* and *`minor`* represent the major and minor numbers for the device (for detailed information see “Major and Minor Numbers” on page 36). Major and minor numbers are dependent upon the configuration of the operating system, as well as any devices and drivers that have been dynamically installed.

For example, to create a node for the second floppy disk that has a major number of `0` and a minor number of `16`, use the following command syntax:

```
# mknod /dev/fd1 b 0 16
```

Whenever a root file system is created, device nodes need to be created within the root file system for various utilities to function properly.

To simplify the process of creating device nodes on a new root file system, it is advisable to save the `/etc/nodetab` file associated with the kernel files that are to be booted with this file system.

Device nodes are listed one per line using the syntax below:

```
name : type : major : minor : perm
```

The `mknod` program reads this file and creates all the device nodes described in it when invoked, as follows:

```
# mknod -a filename
```

With the optional *`filename`* following the `-a` switch, the given `nodetab` file is read instead of `/etc/nodetab`.

NOTE: The `nodetab` file is created in `/sys/bsp*` and copied to the `/etc` directory. In effect, there is no difference between the “`nodetab`” and the `/etc/nodetab` files.

LynxOS Device Node Naming Conventions

Like UNIX, LynxOS uses device nodes to access drivers for such devices as floppy disks, hard disks, RAM disks, CD-ROMs and tapes. By LynxOS convention, an individual device node for each device *must* reside in the `/dev` directory.

Additionally, LynuxWorks has designed a standardized naming convention for device nodes stored in `/dev`. Adherence to these conventions is required for proper identification by **`mkboot`** and kernel booting by **`preboot`** to take place.

This section describes the LynxOS device node naming conventions.

NOTE: While reading through this section, the following device support factors need to be considered:

LynxOS supports the following devices on the following platforms:

- IDE devices on x86, PowerPC, *except* as noted below
- SCSI devices on x86 and PowerPC *only, except* as noted below

LynxOS does not support the following devices on the following platforms:

- IDE tape drives on any platform

Floppy Device Naming Convention

LynxOS supports standard floppy disk drives for x86 and PowerPC.

LynxOS also supports TEAC SCSI floppy disk drives for PowerPC *only.*

NOTE: To correctly follow the LynxOS floppy device naming conventions, it is necessary to identify each floppy device type (standard or SCSI), and its data capacity in bytes.

A LynxOS device node name is a character string made up of several fields corresponding to data about the given device.

A device node name beginning with the letter **`r`** indicates that the associated device is handled by LynxOS through a character (“raw”) interface.

Conversely, a LynxOS device node name *not* beginning with the letter `r` indicates that the associated device is handled by LynxOS through a block interface.

NOTE: There are two exceptions to the **r** and no**-r** device node naming convention stated above:

- RAM disk device node names have an **rd** prefix; the associated devices are handled by LynxOS through a block interface.
- No-rewind tape device names have an **nrst** (no-rewind SCSI tape drive) prefix; the associated devices are handled by LynxOS through a character ("raw") interface.

For more information on devices, see *Writing Device Drivers for LynxOS*.

Standard Floppy Device Naming Convention

LynxOS supports standard floppy devices on all x86 and PowerPC systems.

The figure below shows the fields that make up the LynxOS standard floppy disk device name. The device node name must be a contiguous character string.

fd indicates that the floppy drive associated with the device node is a standard floppy drive. This is followed by a *numeric_string* indicating the floppy device's capacity in bytes (value supplied by user). This is followed by a period (.) and a single numeral from 0 - 3 indicating the user-selected floppy drive ID:

Floppy drive indicator	Device capacity (in bytes)	period	Floppy drive ID
fd	*numeric_string*	.	0-3

Figure 3-1: LynxOS Standard Floppy Device Naming Convention

On systems with only one standard floppy drive designated as the boot disk drive, the drive number must be 0.

On systems with more than one standard floppy drive, one of which is designated as the boot disk drive, any drive number from 0 - 3 can be used for any of the floppy drives.

On systems with more than one standard floppy drive, none of which has been designated as the boot disk drive, any drive number from `0 - 3` can be used for any of the floppy drives.

NOTE: For MS-DOS, drive number `0` corresponds to drive `A:` and drive number `1` to drive `B:`.

Using the conventions illustrated above, the device node associated with a block, 3.5 inch, high-density (1.44 MB) standard floppy disk drive designated as drive `0`, is **`fd1440.0`**.

Similarly, the LynxOS node associated with the character interface for a 3.5 inch high-density (1.44 MB) standard floppy disk drive designated as drive `0`, is **`rfd1440.0`**.

SCSI Floppy Device Naming Convention

LynxOS supports TEAC SCSI floppy devices for PowerPC systems only.

The figure below shows the fields that make up the LynxOS SCSI floppy device name. The device node must be a contiguous character string when entered by users.

`fdscsi` indicates that the floppy disk associated with the device node is a SCSI disk. This is followed by a *`numeric_string`* indicating the floppy device's capacity in bytes (value to be supplied by the user). This is followed by a period (`.`) and a single numeral from `0 - 3` indicating the user-selected floppy drive ID number:

SCSI floppy drive indicator	Media capacity (in bytes)	period	Floppy drive ID
`fdscsi`	*`numeric_string`*	`.`	`0-3`

Figure 3-2: LynxOS SCSI Floppy Device Naming Convention

On systems with only one SCSI floppy drive designated as the boot disk drive, the drive number must be `0`.

On systems with more than one SCSI floppy disk drive, one of which has been designated as the boot disk drive, any drive number from `0 - 3` can be used for any of the floppy drives.

On systems with more than one SCSI floppy drive, none of which has been designated as the boot disk drive, any drive number from `0 - 3` can be used for any of the floppy drives.

Using the conventions illustrated in the figure above, the LynxOS device node associated with the block interface to a 3.5 inch, high-density (1.44 MB) SCSI floppy drive designated as drive `0`, is **`fdscsi1440.0`**.

Similarly, the LynxOS node associated with the character ("raw") interface to a 3.5 inch, high-density (1.44 MB) SCSI floppy drive designated as drive `0`, is **`rfdscsi1440.0`**.

Hard Disk Device Naming Convention

LynxOS supports IDE hard disk devices for x86 and PowerPC.

LynxOS also supports SCSI hard disk devices for x86 and PowerPC *only*.

NOTE: To correctly follow the LynxOS hard disk device naming convention, it is necessary to identify the hard disk type (IDE or SCSI).

IDE Hard Disk Device Naming Convention

LynxOS supports IDE hard disk devices for x86 and PowerPC.

A LynxOS device name is a contiguous character string made up of several fields that correspond to relevant data about the associated device.

The following figure details the fields that make up the LynxOS IDE hard disk device name.

The prefix **`ide`** indicates that the hard disk associated with the device node is an IDE device. A period follows the IDE hard disk indicator.

LynxOS supports up to two IDE channels on a single system, with two device positions on each of these channels, for a total of four IDE drives.

The hard drive ID follows the period. For the x86 architecture, the four IDE hard disk drives have the following IDs:

- `0`--primary master
- `1`--primary slave
- `2`--secondary master
- `3`--secondary slave

The partition ID number follows the hard disk ID. LynxOS supports up to 15 partitions per IDE disk, with each partition assigned an ID of `a - o`.

NOTE: On systems with partitioned hard disks, LynxOS treats each partition as a separate device. Each partition requires a separate device node with a partition ID and a hard disk ID.

The figure below details the LynxOS IDE hard disk device naming convention:

IDE disk identifier	Period	Hard disk ID	Partition ID
`ide`	`.`	`0-3`	`a-o`

Figure 3-3: LynxOS IDE Hard Disk Drive Naming Convention

On systems with only one IDE hard disk that has been designated as the boot disk drive, the drive number must be `0`.

On systems with more than one IDE hard disk, one of which has been designated as the boot disk drive, any number from `0 - 3` can be used for any IDE disk.

On systems with more than one IDE hard disk, none of which has been designated as the boot disk drive, any number from `0 - 3` can be used for any IDE disk.

Using the conventions above, the LynxOS device node associated with the block interface to a primary master IDE device is **`ide.0`**.

Similarly, the LynxOS device node associated with the character interface to a primary master IDE device is **`ride.0`**.

The device node associated with the block interface to a partition **`a`** on a primary master block IDE device is **`ide.0a`**.

The device node associated with the character interface to partition **`a`** on a primary master IDE device is **`ride.0a`**.

SCSI Hard Disk Device Naming Convention

LynxOS supports SCSI hard disk devices for x86 and PowerPC only.

A LynxOS device node is a contiguous character string made up of several fields that correspond to relevant data about the associated device.

The following figure details the various fields that make up the LynxOS SCSI disk name.

In the LynxOS SCSI device naming conventions, the prefix **sd** indicates that the device node is associated with a SCSI device. The SCSI controller ID follows the **sd** prefix, which is then followed by a period (.).

LynxOS supports up to a total of 16 SCSI devices: 4 devices per each of 2 channels on narrow SCSI architecture; 8 devices per each of 2 channels on wide SCSI architecture. A device ID of `0 - 7` for narrow, or `0 - 15` for wide SCSI architecture needs to be assigned by the user. The device ID follows the period.

LynxOS supports up to 15 partitions per hard disk; a partition ID of `a - o` identifies a partition on the SCSI disk.

NOTE: LynxOS treats each partition as a separate device. Each partitions requires a separate special device file with a partition ID and a controller ID.

The figure below details the LynxOS SCSI drive naming conventions:

SCSI disk identifier	controller ID	period	Device ID	Partition ID
sd	*alpha_num_string*	.	0-7 0-15	a-o

Figure 3-4: LynxOS SCSI Hard Disk Device Naming Convention

Using the conventions above, for the block interface to a SCSI hard disk connected to an Adaptec 1542 controller, designated as drive number `0`, the LynxOS device node is **`sd1542.0`**.

For the character interface to a SCSI hard disk, connected to an Adaptec 1542 controller, designated as drive number `0`, the LynxOS device node is **`rsd1542.0`**.

For the block interface to partition **a** on a SCSI hard disk, connected to an Adaptec 1542 controller designated as drive number `0`, the LynxOS device node is **`sd1542.0a`**.

For the character interface to a partition **a** on a SCSI hard disk, connected to an Adaptec 1542 controller, designated as drive number `0`, the LynxOS device node is **`rsd1542.0a`**.

NOTE: For x86 systems, if the SCSI hard disk is the only drive (no CD-ROM, tape, RAM disk), or the SCSI disk is designated as the boot disk drive, then the hard disk BIOS configuration must be set to `Not Installed`.

Examples of LynxOS Hard Disk and Partition Device Nodes

The following table below shows examples of LynxOS hard disk and partition device nodes:

Table 3-2: Hard Disk and Hard Disk Partition Device Node Names

Hard Disk Type	Character ("raw") or Block	Controller Name	Hard Disk ID	Partition ID	Hard Disk or Partition Device Node
IDE	Block	N/A	`0`	`a`	`ide.0a`
IDE	Character ("raw")	N/A	`0`	`a`	`ride.0a`
SCSI	Block	Adaptec 1542	`0`	None	`sd1542.0`
SCSI	Character ("raw")	Adaptec 1542	`0`	None	`rsd1542.0`
SCSI	Block	Adaptec 2940	`1`	`c`	`sd2940.1c`
SCSI	Character ("raw")	Adaptec 2940	`1`	`c`	`rsd2940.1c`
SCSI	Block	NCR	`6`	`a`	`sdncr.6a`
SCSI	Character ("raw")	NCR	`6`	`a`	`rsdncr.6a`

Hard Disk Device Node Naming Exceptions

Iomega Jaz and Zip drives have a unique electronic copy protection mechanism. The **`iomega`** utility is used to manipulate the privileges on these devices. Other than the privilege modification tool, Zip and Jaz drives behave as direct-access SCSI devices; see the `iomega` man page for more information.

CD-ROM Device Naming Convention

LynxOS supports IDE CD-ROM devices for x86 and PowerPC.

LynxOS also supports SCSI CD-ROM devices for x86 and PowerPC only.

NOTE: To correctly follow the LynxOS CD-ROM device node naming convention, it is necessary to identify the CD-ROM type (IDE or SCSI).

The subsections that follow detail the LynxOS device node naming conventions for IDE and SCSI CD-ROM devices.

IDE CD-ROM Drive Device Node Naming Conventions

LynxOS supports IDE CD-ROM devices for x86 and PowerPC.

A LynxOS device name is a contiguous character string made up of several fields that correspond to relevant data about the associated device.

The following figure details the various fields that make up the LynxOS IDE CD-ROM device name.

The prefix **`ide`** indicates that the hard disk drive associated with the device node is an IDE device. A period follows the CD-ROM indicator.

The CD-ROM ID follows the period. LynxOS supports up to two IDE channels on a single system, with two device positions on each channel, for a total of four IDE CD-ROM drives per system, requiring a CD-ROM ID of `0 - 3`.

IDE CD-ROM indicator	period	CD-ROM ID
`ide`	`.`	`0-3`

Figure 3-5: LynxOS IDE CD-ROM Device Naming Convention

On systems with only one IDE CD-ROM drive that has been designated as the boot drive, the drive number must be `0`.

On systems with more than one IDE CD-ROM drive, one of which has been designated as the boot drive, any drive number from `0 - 3` can be used for any of the IDE CD-ROM drives.

On systems with more than one IDE CD-ROM drive, none of which has been designated as the boot drive, any drive number from `0 - 3` can be used for any of the IDE CD-ROM drives.

Using the conventions above, the device node associated with the block interface to an IDE CD-ROM designated as drive number `0` is **`ide.0`**.

The device node name associated with the character interface to an IDE CD-ROM designated as drive number `0` is **`ride.0`**.

SCSI CD-ROM Device Naming Convention

LynxOS supports SCSI CD-ROM drive devices for x86 and PowerPC *only*.

A LynxOS device node is a contiguous character string made up of several fields that correspond to relevant data about the associated device.

The following figure details the various fields that make up the LynxOS SCSI CD-ROM device node.

The prefix **sd** indicates that the CD-ROM drive with which the device node is associated is a SCSI device. This is followed by the CD-ROM controller ID, followed in turn by a period.

The CD-ROM device ID follows the period. LynxOS supports up to a total of 16 SCSI CD-ROM devices: 4 devices per each of 2 channels on narrow SCSI architecture; 8 devices per each of 2 channels on wide SCSI architecture. A device ID of `0 - 7` for narrow, or `0 - 15` for wide SCSI architecture needs to be assigned by the user.

SCSI CD-ROM indicator	Controller ID	period	Device ID
sd	*alpha_num_string*	.	0-7 0-15

Figure 3-6: LynxOS SCSI CD-ROM Device Node Naming Convention

Using the conventions above, for a block SCSI CD-ROM drive connected to an Adaptec 1542 controller designated as drive `0`, the device node is **`sd1542.0`**.

For a character SCSI CD-ROM drive connected to an Adaptec 1542 controller designated as drive `0`, the device node is **`rsd1542.0`**.

Examples of LynxOS CD-ROM Device Nodes

The table below shows example CD-ROM device nodes:

Table 3-3: Examples of CD-ROM Drive Device Node Names

CD-ROM Drive Type	Character (“raw”) or Block	Controller Name	CD-ROM Drive ID	CD-ROM Device Node
IDE	Block	n/a	`0`	`ide.0`
IDE	Character (“raw”)	n/a	`0`	`ride.0`
SCSI	Block	Adaptec 1542	`0`	`sd1542.0`
SCSI	Character (“raw”)	Adaptec 1542	`0`	`rsd1542.0`

Table 3-3: Examples of CD-ROM Drive Device Node Names (Continued)

CD-ROM Drive Type	Character (“raw”) or Block	Controller Name	CD-ROM Drive ID	CD-ROM Device Node
SCSI	Block	Adaptec 2940	`1`	`sd2940.1`
SCSI	Character (“raw”)	Adaptec 2940	`1`	`rsd2940.1`
SCSI	Block	NCR	`6`	`sdncr.6`
SCSI	Character (“raw”)	NCR	`6`	`rsdncr.6`

Tape Device Naming Conventions

LynxOS supports SCSI tape devices for x86 and PowerPC only.

LynxOS does not support IDE tape devices.

A LynxOS device node is a contiguous character string made up of several fields that correspond to relevant data about the associated device.

Tape drives can be either “rewind-on-open” or “no-rewind-on-open.” Accordingly, device nodes begin with either of the following identifier prefixes:

- **`rst`**--rewind SCSI tape drive
- **`nrst`**--no-rewind SCSI tape drive

A tape drive controller ID follows the **`rst`** or **`nrst`** prefix, followed by a period.

A SCSI tape drive ID follows the period. LynxOS supports up to a total of 16 SCSI tape devices: 4 devices per each of 2 channels on narrow SCSI architecture; 8 devices per each of 2 channels on wide SCSI architecture. A device ID of `0 - 7` for narrow, or `0 - 15` for wide SCSI architecture must be assigned by the user.

The following figure details the various fields that make up the LynxOS SCSI tape device name:

no/rewind SCSI tape drive	Controller ID	period	Device ID
`nrst` `rst`	*`alpha_num_string`*	`.`	`0-7` `0-15`

Figure 3-7: LynxOS SCSI Tape Drive Naming Convention

Using the conventions above, the device node associated with a rewind-on-open SCSI tape drive connected to an Adaptec 1542 SCSI controller is **`rst1542.0`**.

The device node associated with a no-rewind-on-open tape drive connected to an Adaptec 1542 SCSI controller is **`nrst1542.0`**.

RAM Disk Naming Convention

LynxOS supports RAM disk virtual disk drives.

A RAM disk is space in memory that simulates a disk drive, i.e., creates a "virtual" disk drive.

The prefix **`rd`** indicates that the device with which the device node is associated is a RAM disk. The prefix is followed by a period.

LynxOS supports up to 16 RAM disks, each capable of containing up to 4 partitions. To accommodate this number, the following scheme is used:

- After the period, a single character numeric ID from `0 - 15`
- After the numeric ID, a single character alphabetic ID from `a - d`

The figure below details the LynxOS RAM disk naming convention:

RAM disk type indicator	Period	Numeric ID	Alphabetic ID
`rd`	`.`	`0-15`	`a-d`

Figure 3-8: LynxOS RAM Disk Device Node Naming Convention

Using the conventions above, the device node associated with the block interface to a RAM disk with numeric and alphabetic designations of `0` and `a`, respectively, is **`rd.0a`**.

The device node associated with the character interface to a RAM disk with numeric and alphabetic designations of `0` and `a`, respectively, is **`rrd.0a`**.

Major and Minor Numbers

Although users and applications use a device name to refer to a device, internally, LynxOS uses a unique numeric identifier for each device that it supports. This identification consists of a major and a minor number. Major and minor numbers are used by the kernel to identify a device. Major numbers represent a general

device type. Minor numbers represent a specific class or subset of a device. The major and minor numbers of devices on any given system depend on how LynxOS is configured. To view the major and minor numbers of the devices on any system, use the command **`ls -l /dev`**. A listing of the `/dev` directory shows the major and minor numbers of each device node. These are located in column three below as two numbers separated by a comma (`,`).

```
$ ls -l /dev

brw-rw-rw-   1 root       2,13   Apr  7 17:15 fd1440.1
brw-------   1 root       1,0    Apr  7 17:15 ide.0
brw-------   1 root       0,32   Apr  7 17:15 sd0b
brw-------   1 root       3,0    Apr  7 17:15 sd1542.0
brw-------   1 root       3,16   Apr  7 17:15 sd1542.0a
```

In the example above, `/dev/fd1440.1`'s major number is `2` and its minor number is `13`.

For additional information on major and minor numbers, see the Chapter "Booting LynxOS" in the *LynxOS Installation Guide*.

Managing Terminals

When using a LynxOS computer for software development, terminals are an important resource. The LynxOS utilities provide several methods to simplify and organize the job of installing new or different terminals, or adding extra serial communication lines.

Most of the work necessary to manage the terminals on a system involves updating the files in the `/etc` directory.

Enabling Ports for Login

Serial ports on a LynxOS system can be set up to be used by terminals and other RS-232 devices, such as printers or modems. Only some of the serial ports should be recognized by **`init`** as being available for interactive use. On these ports, `init` maintains an active process, usually `login`. Serial ports can be listed in the `/etc/ttys` file. Each line of this file describes a single port as follows:

```
device : flag : config : terminal : login
```

The *`device`* field names a node for a serial port. The port is enabled for *`login`* (actually, enabled for consideration by `init`) if the *`flag`* is non-zero. The

config and *terminal* fields select a configuration and terminal type from the files /etc/tconfig and /etc/termcap.

NOTE: It is convenient to include all serial ports in this file, even those unlikely to ever be connected to an interactive terminal. As long as the flag field for these ports is 0, they are not used by init.

For each port enabled in /etc/ttys, init starts a program named by the *login* field of the description line for the port. The standard input, standard output, and standard error file descriptors (0, 1, and 2) are connected to the port. For interactive software development, the *login* program is /bin/login, although any program can be substituted (for example, a simpler version of login that ignores passwords).

init monitors the activity of all the child processes it creates.When any of these child processes terminates, init creates another child process to take its place. Pseudo-tty devices used by network and window-management software are also listed in the /etc/ttys file. They are always disabled. The entries in the file are used to bypass login and provide terminal type information when login is started by a network daemon.

Describing Terminals

Before using a text editor or running utilities, Users must understand the actual terminal device being used before editing files or running utilities. LynxOS stores terminal characteristics and capabilities in the /etc/termcap database file.

Each different make or model of terminal used on a system should be included in the database. The details of a terminal description are rather involved; therefore, a list of only common terminal capabilities is associated with corresponding strings and numeric values. For more information on terminal descriptions, see the **termcap** file format reference pages.

The terminal type can be specified in a relatively static manner as the *terminal* field in the /etc/ttys description of the port. If the terminal type for the port varies, as it could on a modem port, the *terminal* field should be dialup or unknown.

The value of this field is placed in the environment by login as the value of the **TERM** environment variable. The TERM variable can be examined and possibly changed by the shell initialization scripts. It is ultimately used by the editor or other programs to determine the terminal type.

Serial Port Configurations

The serial port driver, called the `tty` driver, controls the details of input and output through a serial port. The following characteristics can be controlled:

- Baud rate
- Input character echoing
- Character mapping of certain characters to driver-supported editing and process management functions

All communication with the driver (as with all drivers) is through the **`ioctl`** system call. Because there are so many `tty` driver features, a program might have to call `ioctl` several times to configure the serial port for its needs. The LynxOS library routine **`ttyconfig`** simplifies this process by allowing the details of the configuration to be kept in the `/etc/tconfig` file.

Using `ttyconfig`, a program simply requests a configuration by name, and the `ioctl` calls are performed automatically. At least one entry, called `default`, should be in the `/etc/tconfig` file. Others may be added, or the file can be forever ignored. However, because `login`, `stty`, and `tset` use the information, the file should exist. For more information on creating configuration entries, see the `ttyconfig` man page.

User Accounts

Typically, each user on a LynxOS system has a unique account. At a minimum, each user account needs the following information set up:

- An entry in the `/etc/passwd` file
- A directory controlled by the user

The user directory is called the home directory of that user. LynuxWorks recommends that system administrators establish file protections such that only the superuser can perform the work of creating accounts (See “File Permissions” on page 42.).

There are two unique attributes of a user:

- User name
- User ID number

Generally, the user name is some adaptation of a person's name. The user ID number is an arbitrarily chosen integer between `0` and `65534`; it is rarely manipulated or even seen. However, because file ownership is recorded in terms of user ID, different users *must* have distinct user ID numbers

---

NOTE: User ID number `0` is reserved for the superuser.

Root and Setup Accounts

The LynxOS **`setup`** system administration utility enables the creation of root (superuser) and setup accounts. To invoke this utility, log in to the LynxOS system as **`setup`**.

System administrators can create `setup` and `root` accounts for the LynxOS machine with the `setup` utility. Typically, `setup` and `root` accounts are reserved for special usage (for example, to manipulate all other user accounts and data), and require superuser privilege.

LynuxWorks recommends that the `setup` and `root` accounts have their own separate passwords. To give the `setup` account a password, enter **`yes`** at the following prompt:

```
# Do you want to give the setup account a password?
```

When ready, enter the password; the characters are not echoed on the screen. This is to ensure that passwords remain secret. The system asks that the password be reentered for integrity.

If the password entered is too short, the system prompts for a longer, less obvious password. The password-length safety mechanism can be overridden, allowing entry of a shorter password, by repeatedly entering the desired, shorter password; after two such efforts, the system accepts the shorter password.

Using the adduser Utility

The system administrator can add user accounts to the system with the **`adduser`** utility. This utility is automatically invoked by the **`setup`** utility. The `adduser` utility creates the following information:

- An entry in the `/etc/passwd` file
- A home directory

To add a user to the system, enter **`yes`** at this prompt:

```
# Do you want to add a user to the system?
```

For each user added, supply the following information:

- User Name
- User ID Number (the default is the first available number)
- User's Group ID Number (the default group ID is `2`, the staff group ID)

  The user can create a new group by modifying the `/etc/group` file.
- Comment

  Typically, the full name of the user is entered in the Comment field.
- User's Home Directory

  This is the user's initial working directory when logging in to the system; it is owned by the user.
- User's Login Shell

  The login shell is the interface between the user and the LynxOS kernel. LynxOS does not copy the `login` or `shell` configuration files to the user's home directory. The system administrator needs to copy these files. The following lists the five LynxOS shells that can be chosen as default login shells:

Table 3-4: LynxOS Login Shells

Shell	Definition
`/bin/csh`	A hard link to `/bin/tcsh`
`/bin/tcsh`	Enhanced Berkeley C shell
`/bin/sh`	A hard link to `/bin/bash`
`/bin/bash`	Free Software Foundation Korn-compatible shell
`/bin/dlsh`	LynuxWorks proprietary shell

The following example shows the entry for a user named Bob Jones:

```
bob::17:3:Bob Jones:/usr/bob:/bin/bash
```

In the example, Bob's user name is **`bob`**, his ID is **`17`**, his group ID is **`3`**, his home directory is **`/usr/bob`**, his default shell is the **`bash`** shell. In this example, the account password remains to be set in the second field, delimited by the two colons

(`::`). If the password was set, the second field would contain an encrypted, but printable, string.

Using the deluser Utility

System administrators can delete user accounts from the system with the **`deluser`** utility. The **`setup`** utility automatically invokes `deluser`, used to delete the following items:

- Specified user's entry from the `/etc/passwd` file
- The user's home directory and subdirectories, if specified

The first prompt displayed when `deluser` is run is as follows:

```
Do you want to delete a user from the system?
```

To remove a user, enter **`yes`**.

NOTE: The first time `setup` is run, the prompt above does not appear.

Before removing the user's home directory, make sure to back up all files that may be needed in the future.

Understanding Security Issues

In order to protect data from unwanted removal or inadvertent tampering, LynxOS provides a means to protect files and programs. Security on a LynxOS system is based on access permissions, allowing a user to read, write, or execute a file.

File Permissions

LynxOS uses standard UNIX file permission commands and bit settings to protect files. Every file or directory contains a set of permissions that define access privileges to the file/directory owner, members of the owning group, and any others. Each of these categories (owner, group, and other), contain permissions that allow (or deny) read access, write access, or execution access. The owner & group associated with a file are displayed with the `ls -l` command. Typically the Owner is the creator of the file (or the last user to touch it.) The Group is the associated group ID of the owner. Anyone who is a part of the group has the listed permission

settings. Anyone who is not the owner, nor a member of the group falls into the Other category.

To display the permissions of a file or directory, use the `ls-l` command. The permissions are displayed in the leftmost columns:

```
bash$ ls -l

drwxrwxr-x 4  tim    pubs   4096  Oct 29 11:55 my_project
-rwxr-xr-x 1  paul   eng    11900 Nov 2  15:39 a.out
-rw-r--r-- 1  paul   eng    215   Nov 1  17:28 cypher.c
Permissions    Owner  Group
```

Figure 3-9: Listing File Permissions

The read (`r`), write (`w`), and execute (`x`) bits are displayed for the owner, group, and other categories. Also, the file (or directory) owner and associated group is displayed. The leading **`d`** or **`-`** character indicate whether it is a file (`-`) or a directory (`d`). The following table provides the permissions breakdown from the example above:

Table 3-5: File Permission Settings

Directory/ File	Owner Permissions	Group Permissions	Other Permissions	Filename
d	rwx	rwx	r-x	my_project
-	rwx	r-x	r-x	a.out
-	rw-	r--	r--	cypher.c

Users have access to the file or directory if a particular bit is set. For example, on the `cypher.c` file, a user in the Other category would have read-only permission because the write and execute bits are not set (`r--`).

Changing Permissions with chmod

The permissions of a file can be changed with the **`chmod`** command:

```
# chmod mode file
```

Where *mode* are the octal values of the permission settings and *file* is the file to change. For example,

```
# chmod 760 cypher.c
```

To change the mode of the file, users must know the octal values of the permission settings they want to use. In the above example, the mode `760` represents the permission settings for the owner (`7`), group(`6`) and other(`0`). These octals are calculated by adding the various permission settings together. The read, write and execute bits are represented by the following values:

- `r=4`
- `w=2`
- `x=1`
- `-=0`

To determine the octal value of the permissions for a file, add the appropriate values together:

Table 3-6: File Permission Octal Values

Permissions	Octal Value	Calculation
`---`	0	0+0+0
`--x`	1	0+0+1
`-w-`	2	0+2+0
`-wx`	3	0+2+1
`r--`	4	4+0+0
`r-x`	5	4+0+1
`rw-`	6	4+2+0
`rwx`	7	4+2+1

To change the permissions of a file to `rwxrw----`, use the `chmod` command, the appropriate octals for each user setting, and the filename:

```
# chmod 760 cypher.c
```

The `chmod` man page contains additional descriptions and options.

Default Permissions

When a file is created, the effective user ID number of the creating process is assigned to the file. The group ID number is inherited from the directory where the file is created.

The mode of the new file is derived by combining the current **`umask`** of the process with the mode requested by the process when it is called open. The `umask` is a number, when expressed in binary, tells which access bits of a requested mode are to be disabled.

For example, if a process attempts to create a file with mode `0666` octal (`110110110` in binary) and the `umask` is currently `0022` octal, then the file is given mode `0644`, as shown below:

Table 3-7: Example umask Requested Number Result

Requested	1	1	0	1	1	0	1	1	0
`umask`	0	0	0	0	1	0	0	1	0
Result	1	1	0	1	0	0	1	0	0

Changing Effective User ID

Sometimes, applications must access files that would normally be protected from access by the user ID under which the application is running. For example, a program that maintains a simple database might allow all users to add records to it, even though the data files are protected against access with ordinary utility programs.

In addition to the basic protection code bits, the mode of an executable file contains flags that can be used to set the effective user and group ID numbers upon execution. With this facility, a program can be configured so that while it is running the effective user or group ID, which is used to determine access privileges, is changed to that of the owner of any protected data.

The **set-user-ID** and **set-group-ID** flags are referred to by the constants **S_ISUID** and **S_ISGID** as defined in the include file sys/stat.h. To set or clear these bits, enter the following commands:

```
# chmod u+s file
# chmod g+s file
```

NOTE: Programs owned by user ID 0, which have the set-user-ID flag enabled, assume most superuser privileges while running. Such programs must be carefully written to prevent unwanted side effects or security violations.

Process Protection

Processes are also protected against interference from processes running under different user ID numbers. Only the owner of a process and the superuser can send signals to, or set the priority of that process.

When a process begins, it is assigned the user ID of the parent process. When a program is loaded, the user ID of the process changes if the **S_ISUID** bit of the program is set. Although a process is protected from direct interference by other processes, it is possible, nevertheless, for a non-privileged process to set its priority high enough to monopolize the processor.

In the context of a real-time application, this feature is expected of a real-time operating system. However, in a development environment, one user could monopolize system resources. The **setpriority** system call allows the root user to set the maximum priority of all non-root users. Thus, users who do not have root access cannot set their priority above that value.

CHAPTER 4 *Disk Space Management*

This chapter describes managing disk space under LynxOS, and procedures for backing up and restoring data with the **`tar`** (tape archive) utility.

Formatting Media

LynxOS supplies the following two utilities for low-level media formatting:

- **`fmtflop`** for floppy disks
- **`fmtscsi`** for SCSI disks

Usually, low-level formatting is needed for floppies only - it is almost never needed for SCSI disks.

LynxOS does not provide any low-level formatting-specific utilities for IDE disks. The **`mkfs`** utility, which creates a file system on a targeted medium, also formats media by performing the following tasks:

- Destroying all data on the targeted medium
- Preparing the medium for read/write access

Low-level formatting of media is generally necessary *only* after read/write access to it has failed. A typical test of read/write accessibility for a medium is to attempt running `mkfs`.

NOTE: Many other systems combine media formatting with file system creation. In LynxOS, however, the `mkfs` utility is required to create an empty file system on LynxOS only after it has been formatted.

Formatting Floppy Disks

Formatting floppy disks with **`fmtflop`** overwrites all old data.

Additionally, the `fmtflop` command can be used to specify the interleave factor and an initial filler value of a floppy using the following syntax:

```
# fmtflop -v -f filler -i interleave device_name
```

The *`interleave`* factor dictates the physical spacing between each block. Optimally, the *`interleave`* is `1`. The *`filler`* value - any value in the range `0 - 255` - is written into every byte of each block. Default values for *`interleave`* and *`filler`* are `1` and `0xf6`.

The filler is a pattern that is used to catch various errors on the medium as it is formatted. The filler occupies every byte of the medium as it is formatted. In the verification phase of formatting, the medium is checked for the filler value. If it does not match the filler value specified, an error is printed. Choosing different fillers and formatting the floppy multiple times ensures integrity.

Usually, the interleave factor is not specified. Larger interleave factors are useful on very slow machines, which take so much time to process each block read from the floppy that they must wait a full rotational delay to read the next block. Placing the next block further away from the current block gives a slow machine the time it needs to process data and avoids the rotational delay penalty.

Typically, `fmtflop` operates silently. The **`-v`** (verbose) option instructs the program to print each track number as it is formatted.

Formatting SCSI Disks

When the **`fmtscsi`** command is run on a SCSI disk, the SCSI disk might simply ignore the command; if the command is ignored, `fmtscsi` immediately returns. Nevertheless, LynxOS provides the `fmtscsi` command syntax:

```
# fmtscsi -F -i interleave device_name
```

As with floppy disks, *`interleave`* should almost always be `1` (the default). `fmtscsi` requires two confirmations before it actually issues the formatting command to the driver.

Configuring Disk Space

Making File Systems

The **mkfs** (make file system) command creates a new file system on a disk or diskette.

CAUTION! The `mkfs` command destroys all existing device data on the specified device upon execution.

For this reason, only superusers should be allowed to use this command.

The syntax for `mkfs` is as follows:

```
# mkfs -b number_bytes /dev/device_name/inodes
```

Setting Block Size

The **-b** option sets the block size of the new file system in number of bytes. The default block size is 512 bytes. The block size is set by entering a value representing the desired number of bytes (*number_bytes* above) as an argument to the `-b` option. Accepted values are `512`, `1024`, `2048`, `4096`, `8192`, `16384`, and `32768`.

The installation utility **installit** uses **-b 2048** when making a file system.

A large number of bytes per block may be appropriate when the majority of files are large. Choose `512` or `1024` if the disk contains mostly small files. This optimizes file storage on the disk. The values `2048` and `4096` are good general purpose values.

For example, the following command creates a 2048 byte block size file system on partition **c** of an Adaptec 2940 SCSI device `0`, where *device_name* is `sd2940.0c`:

```
# mkfs -b 2048 /dev/sd2940.0c
```

Setting Inodes

The **inodes** argument allows the total number of inodes for a file system to be specified. If `inodes` is not specified, `mkfs` creates one inode for every 16

512-byte block on the device. If the file system is expected to have many small files, a larger value for inodes should be specified. For example, the following command creates a file system with 8000 inodes on partition **b** of an Adaptec 2940 SCSI device 2, where *device_name* is sd2940:

```
# mkfs /dev/sd2940 8000
```

For floppy disks, however, the space used for unnecessary inodes can be important; it may be preferable to choose fewer inodes, as follows:

```
# mkfs /dev/fd1440.0 4
```

After it has been created, the file system's parameters can be examined with the **df** command, for example, as follows:

```
# df -i device_name
```

df shows the number of free blocks available for data after taking into account the blocks used for the inode table and the super block. Once a file system has been created, the device containing it can be mounted with the **mount** command.

Organizing Files

The primary means of organizing files in a LynxOS file system is through a hierarchical directory structure. A directory is a file that records pairs of file names and inode numbers. Additional directories can be included among the files recorded in a directory.

Creating Directories

Upon initial LynxOS installation, the **mkfs** utility is used to establish a root directory. Additional directories are then created below the root directory to establish a working root file system.

The **mkdir** command creates directories with the following command syntax:

```
# mkdir name
```

LynxOS creates each directory with the two special entries: *dot* (.) and *dot dot* (..), which represent the inode of the directory itself and the inode of its parent directory, respectively.

In addition to system operation, directories must also be created for software development or applications. Typically, a directory is created for each individual who has an account on the system, either directly beneath the root, or in some subdirectory (/usr is commonly used for this purpose.).

Additional directories owned by the superuser or by any other users can be added as needed while the system is being used.

Removing Directories

Directories can be removed with the following commands:

- `rmdir`
- `rm -r`

These commands are available to all LynxOS users, not just the superuser. The permissions on certain directories can be set to prevent regular users from removing them.

The **`rmdir`** command removes a specified *`directory`*:

```
# rmdir directory
```

To remove a directory and its contents - files and subdirectories - recursively, use the **`rm -r`** command:

```
# rm -r directory
```

The **`-r`** argument instructs **`rm`** to recursively remove the entire hierarchy beneath *`directory`*, including *`directory`* itself.

CAUTION! The effects of `rm -r` are irreversible. `rm -r` removes all existing directory files and sub-directories associated with each of the arguments. It is advisable to check what directory the command is being executed from before executing it.

Setting up an alias for `rm`, such as `rm -i` in the `.cshrc` or `.profile` file so the operating system issues a prompt before it recursively removes files or directories is also advisable.

Managing Disk Space Usage

Although LynxOS provides no disk space quota mechanism, administrators can arbitrarily exercise tight control over resource consumption by periodically checking each user's space usage.

The du Command

The **du** command reports space consumed under a particular directory. Assuming user accounts are all established with home directories in the `/usr` directory, the following **du** command reports consumption per user:

```
# du -s /usr/*
```

The values reported by **du** are in terms of kilobytes (KB). Note that **du** can be applied to any portion of the directory hierarchy, not just user-owned directories. The total amount of space available in a file system is stored in the super block.

Using df Command

The **df** command reads the super block and reports the free disk space, and, optionally, the number of free inodes.

df reports free space for each file system listed recursively within a super block. Specific file systems can also be specified on the command line.

Using the find Command to Determine File Usage

When disk space usage is over 90 percent, users should delete or back-up any unused files in order to increase storage resources. To determine which files might be eligible for deletion or back-up, use the **find** command to list files that have not been accessed within a specified amount of time. The `find` command uses the syntax below:

```
# find / \( -atime +30 -o -mtime +30 \)
```

The expressions `-atime +30` and `-mtime +30` in the example above tell the `find` command to report all files not read or modified in more than 30 days. The `-atime` value can be set by the user to any number of days.

With a similar command, users can first print a list of files to screen, then remove them using the `find` and `rm` commands as follows:

```
# find / \tmp (-atime +30 -o -mtime +30\) \
-print -exec rm -f {} \;
```

NOTE: Please exercise caution and discretion when using this method; once files are deleted, they are either lost or can only be recovered from a backup system.

Backing Up the System

Regardless of the physical reliability of hard disks, many opportunities exist for data loss due to software errors. For example, executing the `find` command could potentially globally delete critical files. Therefore, a backup system is highly recommended. LynuxWorks recommends the standard UNIX **`tar`** (tape archive) utility.

The tar Command

To back up data on the system, use the **`tar`** command. This command gathers subtrees of the directory structure (or even the entire directory tree) into one file, usually onto some removable device.

Creating the tar Backup File

Once a tar backup file has been created, users can examine the file and selectively extract portions of the saved directory tree with `tar`.

For example, to copy the entire contents of the `/usr/` directory subtree to a system's tape drive, the following command would be used:

```
# tar -cv /usr
```

The **`-cv`** argument directs `tar` to create (**`c`**) a new output file, rather than skip to the end of an existing file, and to display a verbose (**`v`**), file-by-file listing of files copied.

As no output file is specified in this example, the `/dev/tar.default` file is used; normally, a system is configured so that the file `/dev/tar.default` is linked to the device name of the tape drive. If `tar` detects that it has run out of space on the output device, it requests that another storage medium be inserted. For `tar` to be able to detect the end of storage on a medium, the raw device interface must be used.

In the example above, the `/usr/` directory was referred to using an absolute path name. The file names in the headers in the tar file reflect the absolute path names. This implies that when files are later extracted, they are forced into the exact position in the overall directory tree they originally had. If, on the other hand, the original `tar` had been performed as shown below, by first changing the working directory to `/usr` and then giving `tar` the path name `.`(period), the files are written with a relative instead of absolute path name:

```
# cd /usr
# tar -cv .
```

The files can then be extracted from the backup file and placed anywhere in the directory structure.

Restoring Backups

Once users have created a tar backup on a tape, another hard disk, or floppy disk, they can restore the saved information in case of data loss using the following command syntax:

```
# tar -xv
```

The **x** option means "extract." All files in the backup volume are copied to the main file system based on the path names in the headers. As each file is extracted, its name and other statistics are printed on screen. Individual files can be extracted by listing the desired path names after the key, as in this example:

```
# tar -xv /usr/m5/mainprog.c /usr/m5/files.c
```

Each file name header that matches any requested name is restored.

Creating Backup Policies and Procedures

All computer systems need the data on their disks to be backed up. However, the frequency of backups depends on each specific system's requirements. Before installing a new version of LynxOS, create a backup on the system. Files can be backed up to the following types of media:

- Floppy disk
- QIC 1/4-inch cartridge tape
- Helical Scan 8-mm tape
- DAT 4-mm tape
- Removable hard disks

Users can use the `find` and `tar` commands to locate files and write to backup media. For example, all files that have been modified in the last week can be backed up with the following commands:

```
# find / -mtime -7 > /tmp/backup.list
# tar -cvf /dev/device 'cat /tmp/backup.list'
```

Where *device* is the device node for an IDE, SCSI or floppy device. See “LynxOS Device Node Naming Conventions” on page 26.

LynuxWorks recommends that the following system configuration files be backed up on a regular basis:

Table 4-1: Configuration Files Recommended for Backup

/.cshrc	/.Xsession
/etc/rc.d/rc	/.bashrc
/.dlshrc	/.exrc
/.lesskey	/.less
/.login	/.logout
/.motifbind	/.mwmrc
/.profile	/.rhosts
/.subroutines	/.twmrc
/.Xdefaults	/.xdm-config
/.xinitrc	/.Xresources
/.Xservers	/etc/exports
/etc/fstab	/etc/group
/etc/hosts	/etc/hosts.equiv
/etc/magic	/etc/passwd
/etc/printcap	/etc/profile
/etc/starttab	/etc/ttys
/etc/resolv.conf	/etc/inetd.conf
/etc/ethers	/etc/networks
/etc/nodetab	/net/rc.network
/sys/lynx.os/CONFIG.TBL	/sys/lynx.os/uparam.h
/sys/cfg	

NOTE: The list in this table is by no means comprehensive. For any individual system, some of the files listed above need not be saved; additionally, users may also want to save other system configuration files not listed above.

CHAPTER 5 *Shared Libraries*

Users may have an application that relies on shared libraries. Or perhaps users are considering using shared libraries for an embedded application.

This chapter describes shared libraries and explains whether they should be used, how they are used, and how they are built.

Overview

A shared library is a collection of routines, organized so that code can be shared among processes; its primary goal is to reduce system and disk memory usage.

LynxOS supports tools to create and use C and C++ dynamic ELF shared libraries on all of its supported platforms.

Shared library development is supported by all native and cross development tools.

The term *shared library* is not entirely accurate when used with the ELF/SVR4 implementation. The correct term is *shared object*, because shared libraries are not archive (`ar`) libraries. They have more in common with partially linked (`ld -r`) object files. Shared objects can be identified by the **`.so`** suffix. This document, however, uses the term shared library when referring to shared objects.

The main difference in the treatment of LynxOS shared libraries and those on other systems at load-time, is that *lazy linking* is not supported for LynxOS shared libraries. The use of *lazy linking* makes application startup faster, but causes problems in real-time systems with deterministic behavior requirements.

Compatibility between different revisions of a library is easy to maintain, as long as the libraries are functionally compatible; applications linked with the library do not need to be relinked in order to use the newer version.

No source code changes are necessary when making a shared library. Object files that are to be part of the shared library must be compiled as Position Independent Code (PIC) using the **`-fPIC -mshared -shared`** compiler flags.

The LynxOS implementation also includes `dlopen()`/`libdl.so` support. This library provides functions that allow users to open, close, and find symbols in an arbitrary shared library, even if the application in use has not been linked with the library. For more details, see the `dlopen()` man page.

Creating Shared Libraries by Default

Some ELF-based systems, such as Linux, create shared objects by default, and make it necessary to provide a special option to the linker to produce statically linked objects. LynxOS takes the reverse approach. In LynxOS 4.0, the linker produces statically linked objects by default. To produce executables that link with shared objects at run-time, you must link with the option `-mshared`.

Single/Multithreaded Applications and Shared Libraries

When an application links with a shared object, it is important to note if the application uses single-threaded processes or mulithreaded processes. Processes that create multiple threads (using the system call `pthread_create`) should only call thread-safe functions. Thread-safe functions are coded in such a way that they work correctly even when they are concurrently executed by more than one thread.

On the other hand, processes that never create new threads can call non-thread safe functions without difficulty. Since non-thread safe functions may be faster than their thread-safe equivalents, users may prefer to use non-thread safe functions whenever possible. The usual practice is to keep thread-safe functions and non-thread safe functions in different libraries. LynxOS follows this practice with the libraries that it provides, and users should consider following this practice with their own libraries.

The libraries provided by LynxOS are kept in directories that distinguish thread-safe from non-thread safe libraries. The following table details the library types and locations:

Table 5-1: LynxOS Library Types & Directories

Library Type	Directory
Thread-safe shared libraries	`/lib/thread/shlib`
Non-thread safe shared libraries	`/lib/shlib`
Thread-safe static libraries	`/lib/thread`
Non-thread safe static libraries	`/lib`

Effects of Using Shared Libraries

In essence, when users modify the contents of a shared library, they modify all the programs that rely on it, unless they never deploy the new library, or only deploy it selectively. Using a shared library instead of an ordinary non-shared archive library may affect the following:

- System Memory Usage
- Disk Space Usage
- Code Maintenance

System Memory Usage

System memory usage is different because multiple processes can share the library code. The amount of system memory saved (or lost) by using a shared library depends on three factors:

- The contents of the shared library
- The set of programs that are typically run on the system
- The load on the system

(See “Determining the Use of Shared Libraries” on page 61 for additional details and several examples).

The following relationships are true in general:

- A shared library consisting of commonly used routines saves more system memory than one that has many rarely used routines.

  Routines and data items in the shared library that are rarely used waste system memory simply by their lack of use as follows:

  - A rarely used routine wastes memory because the entire shared library resides in memory whenever it is used, regardless of which subset of routines from the library are actually linked.
  - A rarely used data item wastes system memory equal to its size multiplied by the number of processes currently using the shared library. This is because each process has its own copy of the data.

- A shared library that is used by a varied and numerous assortment of programs saves more system memory than a shared library that is used by only one or two programs.

  Multiple concurrent executions of the same program use the same amount of text memory as does a single execution. The operating system ensures that the executing text segments of the same program are shared.

Disk Space Usage

Disk space usage is typically lower because a program linked with a shared library is almost always smaller than the same program linked with an equivalent archive library. The shared library itself is not a factor in disk space usage because it is comparable in size to an equivalent archive library.

Consider this simple “hello world” program:

```
#include <stdio.h>
main()
{
        printf("Hello, world!\n");
}
```

The `printf()` function is included in the C library. If the program is linked to the static C library, the resulting program executable file is close to 36 kilobytes:

```
# gcc -o hellostatic hello.c
# ls -l hellostatic
 -rwxr-xr-x   1 root 36014 Jan 24 01:01 hellostatic*
```

If the same program is linked with the shared C library, then it is only about 5 kilobytes:

```
# gcc -mshared -o hellodynamic hello.c
# ls -l hellodynamic
 -rwxr-xr-x   1 root 5133 Jan 24 01:02 hellodynamic*
```

In many cases, using shared libraries provides a considerable difference in size compared to static libraries.

Code Maintenance

Maintaining programs that use a shared library is somewhat easier than maintaining programs that use an archive library because if an error in the library is fixed, only the shared library needs to be remade. The programs do not need to be re-linked with the library because the operating system maps the shared library code into their address spaces at run time. In effect, modifying the contents of a shared library modifies all of the programs that use it.

NOTE: The above statements hold true only when the change to the shared library is backward compatible. Compatibility problems can be avoided as long as the shared library can be built using the guidelines described later in this chapter. For bug-fixes, compatibility is often automatic. For changes that are broader in scope, adherence to the guidelines may become challenging or impractical.

Determining the Use of Shared Libraries

In order to decide whether or not to use shared libraries, users must take the following factors into consideration:

- The number of different applications that are to run concurrently on the target system
- The percentage of the library code that applications use
- The size of the library's `data` and `bss` sections
- The amount of RAM available on the target system
- The amount of disk, flash, or ROM space available on the target system
- The ease of updating and fixing bugs on shared libraries
- The possibility of performance degradation if using shared libraries

If a target system is running a single application, multiple instances of a single application, or multiple different applications (not concurrently), the using of

shared libraries will probably increase memory and disk space usage. If, however, the target system is running many different applications at the same time (and they use large portions of the same shared libraries), there may be significant reductions in memory and disk space usage.

For comparison purposes, the space requirements of three sets of applications are explored in three pairs of tables that follow. All applications in a given set require a common library.

In the tables “Space Requirements for a 1 MB Library (X) Used by 6 Applications,” “Space Requirements with a 1 MB Library (Y) Used by 6 Applications,” and “Space Requirements with a 1 MB Library (Z) that Includes Data,” space requirements are shown when the associated library is implemented as a non-shared library.

In the tables “Space Requirements for 1 MB Shared Library (X) Used by 6 Applications,” “Space Requirements with a 1 MB Shared Library Used by 6 Applications,” and “Space Requirements with a 1 MB Library (Z) that Includes Data,” space requirements are shown when the associated library is implemented as a shared library.

Example 1

Each of the applications shown in the next two tables uses half of a 1 MB library, but uses a different mixture of the library’s routines. Where these applications do not use a shared library, they require 3 MB of RAM and disk space. When they employ a shared library, they require only 1 MB of RAM and disk space.

Table 5-2: Space Requirements for a 1 MB Library (X) Used by 6 Applications

	Library X Usage • library text used by application										**Space Requirements for Library X on Target**	
	text										**RAM**	**Disk**
Static Library X											0	0
Application A	•	•	•	•	•						.5 MB	.5 MB
Application B		•	•	•	•	•					.5 MB	.5 MB
Application C			•	•	•	•	•				.5 MB	.5 MB
Application D				•	•	•	•	•			.5 MB	.5 MB
Application E					•	•	•	•	•		.5 MB	.5 MB
Application F						•	•	•	•	•	.5 MB	.5 MB
TOTALS											3 MB	3 MB

Table 5-3: Space Requirements for 1 MB Shared Library (X) Used by 6 Applications

	Library X Usage • library X deploys element + library text used by application										**Space Requirements for Library X on Target**	
	text										**RAM**	**Disk**
Shared Library X	•	•	•	•	•	•	•	•	•	•	1 MB	1 MB
Application A	+	+	+	+	+						0	0
Application B		+	+	+	+	+					0	0
Application C			+	+	+	+	+				0	0
Application D				+	+	+	+	+			0	0
Application E					+	+	+	+	+		0	0
Application F						+	+	+	+	+	0	0
TOTALS											1 MB	1 MB

Example 2

Each application shown in the next two tables uses 10% of a 1 MB shared library. Each application uses a different portion of the library, all mutually disjoint. Where these applications do not use a shared library, they require 0.6 MB of RAM and disk space (with all six applications running). When using a shared library, they require 1 MB of RAM and disk space.

This is a worst-case scenario; it is not possible to save any space by using a shared library for this mix of applications. This example is for illustration purposes only. It is unlikely that a group of applications would use completely disjoint sets of library routines.

NOTE: If the unused routines were removed from the shared library, the memory usage for both shared and non-shared libraries would be the same.

Table 5-4: Space Requirements with a 1 MB Library (Y) Used by 6 Applications

	Library Y Usage • library text used by application										**Space Requirements for Library Y on Target**	
	text										**RAM**	**Disk**
Static Library Y											0	0
Application A	•										.1 MB	.1 MB
Application B		•									.1 MB	.1 MB
Application C			•								.1 MB	.1 MB
Application D				•							.1 MB	.1 MB
Application E					•						.1 MB	.1 MB
Application F						•					.1 MB	.1 MB
TOTALS											.6 MB	.6 MB

Table 5-5: Space Requirements with a 1 MB Shared Library Used by 6 Applications

	Library Y Usage • library X deploys element + library text used by application										**Space Requirements for Library Y on Target**	
	text										**RAM**	**Disk**
Shared Library Y	•	•	•	•	•	•	•	•	•	•	1 MB	1 MB
Application A	+										0	0
Application B		+									0	0
Application C			+								0	0
Application D				+							0	0
Application E					+						0	0
Application F						+					0	0
TOTALS											1 MB	1 MB

The preceding examples assume that the library contains no `data` or `bss` sections. While this is the best way to build a shared library, it is not always possible to do so.

Due to the inability of applications to share data, the `data` and `bss` sections of a shared library are not treated the same way as the `text` section: Every process gets a full copy of the `data` and `bss` sections whether it uses all of it or not.

Example 3

In the applications shown in the next two tables, the issues surrounding `bss` and `data` section utilization in a shared library are illustrated. Both shared and non-shared versions of the same library contain 0.5 MB of `text`, 0.3 MB of `data`, and 0.2 MB of `bss`. Each application uses forty percent of the `text` and forty percent of the `data` and `bss` sections.

In the case of the non-shared library (the table below), memory usage is 2.4 MB and disk usage is 1.9 MB. In the case of the shared library (the second table), the memory usage has increased to 3.5 MB, while the disk usage has dropped to 0.8 MB.

Table 5-6: Space Requirements with a 1 MB Library (Z) that Includes Data

Type	Library Z Usage • library text/data/bss used by application									Space Requirements for Library Z on Target	
	text					data			bss	RAM	Disk
Static Library Z										0	0
Application A	•	•				•	•			.4 MB	.4 MB
Application B		•	•				•	•		.4 MB	.4 MB
Application C			•	•				•	•	.4 MB	.3 MB
Application D				•	•				•	.4 MB	.2 MB
Application E	•				•	•				.4 MB	.3 MB
Application F		•		•			•		•	.4 MB	.3 MB
TOTALS										2.4 MB	1.9 MB

Table 5-7: Space Requirements with a 1 MB Library (Z) that Includes Data

Type	Library Z Usage • library text/data/bss + library data/bss used by application - library data/bss allocated but not used by application * library text used by application										Space Requirements for Library Z on Target	
	text					data			bss		RAM	Disk
Shared Library Z	•	•	•	•	•	• 1	• 1	• 1	• 2	• 2	.5 MB	.8 MB
Application A	*	*				+	+	-	-	-	.5 MB	0
Application B		*	*			-	+	+	-	-	.5 MB	0
Application C			*	*		-	-	+	+	-	.5 MB	0
Application D				*	*	-	-	-	+	+	.5 MB	0
Application E	*				*	+	-	-	-	+	.5 MB	0
Application F		*		*		-	+	-	+	-	.5 MB	0
TOTALS											3.5 MB	.8 MB

1. Shared library occupies disk space only.
2. Shared library bss does not occupy disk space.

As the preceding examples show, a well designed shared library may provide a significant savings in the memory space needed. In an embedded system with tight memory constraints, these savings could mean the following:

- A given application now fits in the space allowed on board.
- RAM can be decreased (cost savings).
- More space for new features

The preceding examples also show that a well designed shared library may provide a significant savings in the disk, flash, or ROM space needed. In an embedded system with tight memory constraints, these savings could mean the following:

- A given application now fits in the space allowed.
- The flash/ROM chip count can be decreased (cost savings).
- The flash/ROM chip size can be decreased (cost savings).

- A disk may not be needed (cost savings).

Choosing Shared Library Contents

Choosing the contents of a shared library involves more considerations than choosing the contents of an ordinary non-shared archive library. It is perhaps the most important factor determining whether or not the shared library decreases system memory usage.

When choosing routines for the shared library, use the following guidelines:

- It is generally a good idea to include routines such as **`printf`** that are used often by many different programs.
- It is generally a bad idea to include routines that are rarely used. Recall that when a shared library is used, it is loaded into memory, regardless of what routines have been linked.
- It is generally a bad idea to include routines that have large amounts of data. Recall that the data of a shared library is not shared, so each process that uses the library gets its own copy of the data. Even if a routine is commonly used, it can still waste space, because any program that does not happen to use it still gets a copy of its data.

How to Save Space

In order to minimize the space used by a shared library, global data should be minimized. One technique for doing this is to use local (stack) variables instead of global variables wherever possible. Another technique is to allocate buffers dynamically (for example, with **`malloc()`**) instead of statically. These are effectively identical methods of reducing the `data` section of the shared library. This is important because each process using the shared library gets its own separate copy of the data section.

The above space-saving techniques have the added benefit of making maintenance of the shared library easier by reducing the number of external symbols in the library. The number of external symbols can be further reduced by explicitly using static variables wherever possible.

Updating Shared Libraries

The way that shared libraries are designed allows for an application to be updated (by rebuilding and replacing the target shared library) without the need to relink the

application. This means that bug-fixes or other library changes can be made in the field without replacing the entire application. This feature of shared libraries may require extra work on the part of the library designer to ensure that a new library is compatible with previous versions.

To support compatibility between newer and older versions of the library, shared libraries use a global offset table to access function calls. This adds one or more additional instructions to the normal function call process. These extra instructions produce slightly longer execution times for library calls. In general, the performance decrease is not measurable.

Libraries Provided

LynxOS provides the following system shared libraries. Single-threaded versions are located in the `/lib/shlib` directory; multi-threaded versions are located in the `/lib/thread/shlib` directory. The X and Motif shared libraries are located in `/usr/lib/shlib`.

Table 5-8: Shared Libraries in the LynxOS Distribution

Library	Description	Availability	
		Single-threaded	Multi-threaded
`libc`	Standard C Library	•	•
`libm`	Standard Math Library	•	•
`libdl`	Dynamic Linker Library	•	•
`libalias`	Packet aliasing library for network address translation and masquerading.	•	•
`libcurses`	CRT Screen Handling Library	•	•
`libbsd`	Network Address Resolution Library	•	•
`libmsng`	LnyxOS Messenger Library	•	•
`libgcc`	GNU C library	•	•
`libstdc++`	GNU C++ Support Library	•	•
`libXt`[1]	X Windows Support Library	•	

Table 5-8: Shared Libraries in the LynxOS Distribution (Continued)

Library	Description	Availability	
		Single-threaded	Multi-threaded
libX11[1]	X Windows Toolkit Library	•	
libICE[1]	X Communications Library	•	
libSM[1]	X Sessions Management Library	•	
libXext[1]	X Extensions Library	•	
libPEX5	Phigs (Programmer's Hierarchical Interactive Graphics System) Extension to X	•	
libXIE	XIE server extension library	•	
libXaw	X Athena widget library	•	
libXi	X input extension	•	
libXmu	X miscellaneous utilities	•	
libXp	X print library	•	
libXpm	X pixmap library	•	
libxtst	X testing extension	•	
libXm[1]	Motif Library	•	

1. x86 and PowerPC only.

Creating Shared Libraries

Shared libraries are easy to create. The following example shows how to create a shared library named `myshared.so` from the file `myshared.c`:

```
# gcc –shared –mshared –fPIC –o myshared.so myshared.c
```

The options in this line are:

`-shared` The object file of this command will be a shared object.

`-mshared` Any references to other library functions will be to those found in the LynxOS-provided shared libraries.

`-fPIC`	The text created will be position independent, i.e., it can be linked to shared objects.
`-o`	The following token, `myshared.so`, is the name of the created shared library.

The option `–shared` is used to create a shared object.

The option `–fPIC` causes the compiler to produce position-independent code. In other words, it produces text that does not contain pointers in its read-only area. Instead, the text works with a jump table so that it can be dynamically linked to functions in the shared libraries, whose locations in memory are not known until run-time. At run-time, the addresses of those functions are determined and stored in the jump table. When the function is called, the call address is fetched from the jump table.

Linking to a Shared Library

In order to link with one of the shared libraries supplied by LynxOS, users must tell the compiler/linker to look for libraries in one of the shared library directories. There are two ways to do this: by specifying the `-mshared` flag; or by specifying the directory to search using the `-L` *`directory`* flag.

The following commands are for single-threaded applications. Either command can be used; both are equivalent:

```
$ gcc -o output_file source_files -mshared
$ gcc -o output_file source_files -L /lib/shlib
```

The `-mshared` option links the executable with the dynamic shared libraries in `/lib/shlib/`.

The following commands are for multi-threaded applications (i.e., if the program calls `pthread_create`). Either command can be used; both are equivalent. The user needs to choose thread-safe binary functions. Use the option `-mthreads` to the `gcc` command.

```
$ gcc -o output_file source_files -mthreads -mshared
$ gcc -o output_file source_files -mthreads -L \
/lib/thread/shlib
```

The use of `-mshared` and `-mthreads` together selects the dynamic shared libraries that are thread-safe, found in `/lib/thread/shlib`.

CHAPTER 6 X & Motif Development Package

This chapter provides an overview of the X & Motif Development Package features, X libraries and utilities, and troubleshooting X issues.

NOTE: The X & Motif Development Package is not included in all LynxOS packages. Only native development systems include X & Motif. For cross development systems, X & Motif can be purchased separately. Contact your LynuxWorks sales representative for more information.

Installing and Starting X

The X & Motif packages are not installed during the standard LynxOS installation. X & Motif requires a separate installation procedure. Follow the instruction provided in the *LynxOS Installation Guide* for detailed information.

Once X is installed, it can be configured with the **`configX`** utility. For detailed instructions, see the *LynxOS Installation Guide*. Once the proper display device settings are configured, start x with the **`startx`** command:

```
# startx
```

X Server Features Overview

The following is a list of features of the X & Motif Development Package:

- VGA and SVGA support
- Multiple display resolutions and color depth (depending on graphics adapter)
- Hot-key switching between display resolutions

- Support for ISA, PCI, and AGP bus graphics adapters
- Server extensions like the SHAPE, MIT-SHM, Multibuffering, DEC XTRAP, XIdle, XInput Extension, and others
- Scalable fonts with font server
- Multiple display screen support
- Touch screen support for some ELO graphics, Carroll Touchscreen, Lucas/Deeco and MicroTouch touchscreens
- Dual touchscreen support
- Standard X11 clients
- Support for various mice, both serial and PS/2-compatible, including the Intellimouse
- Example files include: `.xinitrc`, `.xsession`, `.Xdefaults`, and `.mwmrc`

X Server Technology from Metro-X

This release of the LynxOS X & Motif Development Package for x86 is based on the Metro-X Enhanced Server Set technology. This server provides the ability to dynamically load adapter modules via a loader, allowing for up-to-date support of current graphics adapters. The adapter modules are in ELF format and are operating system-independent libraries. Also, the loader architecture is easily extensible to support other formats and even other processor architectures.

Networking and the X Server

The X11 graphics system uses a client-server model. The X server is configured to the local system's graphics adapter and controls the output to the local display. The X server can also accept connections from other programs (clients). These X clients instruct the X server on what to draw on the graphics display. The program **`xterm`** (the terminal emulator) is a sample X client. These X clients can connect to the server using networking protocols such as TCP/IP, or UNIX domain sockets (on a local system).

TCP/IP is used for connections between remote client and server systems.The UNIX domain connections, are used for connections between the client and server from the same machine or system. The UNIX domain connection configured as a local connection is faster than a TCP/IP connection.

When the X server starts, it creates a TCP/IP listening socket to accept connections from X clients. It then makes a UNIX domain listening endpoint for local connections. If both connection attempts fail, the server aborts. Both listening end points can be active at the same time.

An X client connects to the X server via the local UNIX domain connection, because typically, this connection is faster. If the UNIX domain connection fails, the X client attempts to make a TCP/IP connection by calling the X server TCP/IP listening on socket 6000.

X Server Hot Key Exit

The server in the X & Motif Development Package supports a Hot Key Exit, useful for terminating the X server. The key combination is **Ctrl-Alt-Backspace**.

Hot Key Resolution Switching

When configuring the X & Motif Development Package, several screen resolutions can be selected for most cards. Users can change between these resolutions on the fly while running X by pressing these key combinations:

- **Ctrl-Alt-+** (plus) (higher resolution)
- **Ctrl-Alt--** (minus) (lower resolution)

Hardware Panning

The X & Motif Development Package supports hardware panning, which allows for resolutions higher than the terminal's physical resolution. Windows can extend beyond the screen's physical boundaries and can be viewed by scrolling into the extended area.

Touchscreen Support

Touchscreen controllers are simple devices that transmit absolute x and y coordinates to the serial driver when the screen is pressed. The X11 server reads these coordinates and converts them into X11 events, and treats the touch screen as a one-button mouse.

International Keyboard Support

Metro-X uses the X Keyboard extension to support various keyboards. The keyboard description contains information about the physical layout of a keyboard, the key codes of the keyboard, and the symbol information needed to map the keycodes into `keysyms`. Metro-X provides many predefined configurations, sorted by language and country. These configurations are selected through the **`configX`** utility. For instructions on using **`configX`**, see the *LynxOS Installation Guide*.

Multiheaded Servers

On x86 systems, the Matrox Productiva G100 Multi-Monitor server option provides a multiheaded X server. A single keyboard and mouse can be used to control up to sixteen screens on separate monitors. This provides the ability to run many applications on a single window. This can be useful for image-processing work.

In a multiple screen configuration there is one X process, one mouse, and one keyboard**,** but several screens. The cursor can be moved between the different screens. Clients connect to the different screens with the syntax `-display :0.`*`X`* where *`X`* is the screen number.

X Libraries

The development system provides access to the standard X and Motif routines and functions via a link library interface. This allows the developer to port existing X and Motif code to LynxOS and to incorporate the X and Motif facilities into new or existing applications.

The Development System

The X & Motif Development Package consists of a set of link libraries, header files and utilities for the development of X and Motif applications.

Library Documentation

Information about the definition, use, and function of each library routine can be found in the X Windows documentation set provided with the product distribution.

An overview of libraries by platform is listed below.

Table 6-1: x86 Libraries

File	Description
`libX11_s.a`	The shared-library version of `libX11.a` is called `libX11_s.a`.
`libXt_s.a`	The shared-library version of `libXt.a` is called `libXt_s.a`.
`libSM.a`	This is an extension of the **`Xt`** library and support for Session Management.
`libSM_s.a`	The shared-library version of `libSM.a`.
`libICE.a`	This is also an extension to the **`Xt`** library and provides support for communications.
`libICE_s.a`	The shared-library version of `libICE.a`.
`libXext_s.a`	The shared-library version of `libXext.a` is called `libXext_s.a`.

x86/PPC Libraries

Table 6-2: x86/PPC Libraries

File	Description
`libX11.a`	The standard MIT X11R6 `Xlib` is `libX11.a`, and provides the lowest level interface between the X application and server. It includes routines like `XOpenDisplay()`, which the client uses to connect to the server. Enhancements have been made to `libX11.a` for increased reliability and faster local server/client connection mechanisms.
`libXau.a`	This is a simple Authorization Protocol for X11.
`libXaw.a`	This is the Athena widget set, which provides building blocks used by many X11R3 and X11R4 clients. It is rarely used by Motif applications.
`libXdmcp.a`	This is the XDMCP (display manager Authorization Protocol) library used by the `xdm` (X display manager) client. The X server has hooks to completely support this protocol.

Table 6-2: x86/PPC Libraries (Continued)

File	Description
libXext.a	This library contains extensions to the X11 Protocol. It includes the SHAPE extension, which makes the round windows used by oclock and xeyes possible. The X server is compiled with the SHAPE extension.
libXmu.a	This library contains miscellaneous routines.
libXtrap.a	This library is an input simulation library. The library provides support for the simulation and testing of keyboards, mice, and other types of input devices, and the redefinition of the operation of those input devices as well.
libXt.a	The X Toolkit, libXt.a, is a set of routines built on top of Xlib that implements the object oriented widget concept. This toolkit is the basis for both the Athena widget set and the Motif widget set. The use of a widget set provides menus and other high-level functions that allow developers to write applications more quickly.

Motif Libraries

Table 6-3: Motif Libraries

File	Platform	Description
libXm_s.a	x86 only	The shared-library version of libXm.a.
libXm.a	x86/PPC	This is the main Motif library. It contains a collection of widgets that implement the OSF/Motif look and feel. Also included are a set of convenient functions to create and manipulate the widgets.
libMrm.a	x86/PPC	This is the Motif resource-manager library. It is used in conjunction with the interface definition language and compiler.
libUil.a	x86/PPC	This library provides support for the User Interface Language compiler.

Other Libraries

Table 6-4: Other Libraries

File	Description
`libFS.a`	Font server library.
`libXi.a`	Provides support for the client side `Xinput`.
`libXtst.a`	This is the `XTest` library.
`libxkbfile.a`	Provides support for the XKeyboard.
`libXxf86dga.a`	This is provides Direct Graphics Interface support.
`libXau.a`	Provides support for X authentication.
`libXdmcp.a`	X Device Management Protocol support.

X Utilities

imake

`imake` is a compiler-independent build facility that is provided for the portability of X clients. It is recommended that users use the `imake` compiler utility when compiling programs from another source (such as public domain clients). The `imake` compiler should be run with the `-I/usr/lib/X11/config` flag as indicated in the `imake` portion of the man pages.

The **`xmkmf`** utility provided, calls the `imake` utility with the appropriate flags and can be used to generate Makefiles from `Imakefiles`.

uil

`uil` is a free-form user interface language provided as part of Motif for the designing of screens and widgets.

Troubleshooting X

Before Contacting LynuxWorks Technical Support

Before contacting LynuxWorks Technical Support for an X related problem, execute the **`metro-x-pr`** script distributed with the X distribution and supply all the information it requires. This generates a report file that contains important system information. Provide this report file to LynuxWorks Technical Support (see "Technical Support" on page xiii).

Similarly, for Motif-related issues, **`metro-motif-pr`** is provided with the Motif distributions.

Modifying Disk Cache Blocks

The performance of an application using the disk subsystem can be greatly affected by the size of the Disk Buffer Cache residing in kernel memory. The disk buffer cache is used to increase the performance of the operating system. The default value of the Disk Buffer Cache is configured for smaller embedded systems. X windows files tend to be large and quickly consume the disk buffer cache. It is recommended that the buffer cache be increased to at least 1024 blocks. Disk caches with a size of 2048 to 4096 blocks are common on systems that heavily use X and/or the file system. The default entry for the buffer cache is provided below.

```
/* number of cache memory blocks */
#define CACHEBLKS 1024
```

This entry from the kernel configuration file `/sys/lynx.os/uparam.h` must be modified to an appropriate value for the specific system configuration. See the *LynxOS User's Guide* for more information.

Limited Colors

Some window managers auto allocate many colors by default. With a limited color server, there may be fewer colors for applications. Reconfigure the window manager to use fewer colors. Refer to your window manager man page for more information.

Unsupported Programs

The `contrib` directory of the X distribution provided by `x.org` is neither supported nor supplied by LynuxWorks.

Some standard X clients are not supported on LynxOS because the underlying kernel or utility support does not exist. Unsupported clients include **`xload`** and **`xmh`**.

Xconsole and newconsole

The **`xconsole`** can be used to view console messages.

LynxOS also provides a facility similar to **`xconsole`** to change the console. See the man page for `newconsole(1)`. Run **`newconsole`** in an xterm window. Root privileges are required to change the console.

CAUTION! If a user changes the console to the pseudo-tty associated with an xterm, that xterm and pseudo-tty must be kept active. Do not close the xterm window. Otherwise, console messages may fill up the `pty` queue and cause the system to lock up. To close the xterm or exit X, run **`newconsole`** to set the console to another device, `/dev/atc0`, for example, before exiting.

Saving Errors

The X server creates an error file `Xerrors`, in the `/usr/lib/X11/Metro` directory. In case of server problems, examine this file to help diagnose the problem.

Window Manager

The X distribution ships with two window managers: **`mwm`** (if the Motif distribution is installed), and **`twm.`** Only one window manager can run at any given time.

Real-Time Priorities and X

By default, the X server and initial clients run at the priority of the process that started them. X servers and clients do not adjust priority. Raising the priority of the server and window manager above that of other applications, shells, and X clients can improve the interaction with the window system.

If demand-paged virtual memory is active on the system, X programs should be allowed to be swapped because they use so much memory. To do this, make sure that **`vmstart`** (typically run from `/bin/rc`) is running at a higher priority than the priorities chosen for the X programs to swap.

Serial Printer

If both the printer and mouse are configured to use COM1 (`/dev/com1`), there may be abnormal behaviors if the printer is left configured. Disable the serial printer port by editing the `/etc/printcap` file.

Diamond Viper 550 TNT video card (x86 only)

Some graphics adapters, for example, the Diamond Viper 550 TNT, require that the SMEMS value in the LynxOS kernel be increased from the default 10 to 50 in the `/sys/lynx.os/uparam.h` file.

If an error returned by the server resembles the following, change and rebuild the kernel.

```
Fatal server error:
xf86MapVidMem: failed to smem_create
(Argument invalid/improper)
```

X Development Troubleshooting

SKDB (Simple Kernel Debugger) and X

While running X, SKDB (Simple Kernel Debugger) can only be run from `COM2` and cannot be run from the console. Refer to the *Total/db User's Guide*, provided with LynxOS distribution, to configure SKDB on `COM2`.

X and Motif Libraries in /usr/lib

The X and Motif libraries are located in `/usr/lib`. Some LynxOS compiler/linker combinations do not, by default, look in `/usr/lib` for libraries. The **`Install.XM`** script creates symbolic links for some of these libraries into the `/lib` directory during the X and Motif installation procedures.

In some cases, users must set the library search path for compilers and linkers (**-L** for **gcc**, or setting LIBPATH, for example). Refer to the compiler and linker documentation for detailed instructions.

X and Threads

X and Motif have been developed in a single-threaded environment and are not considered "thread safe." A multithreaded application may use X and Motif library calls if all the calls are confined to a single thread or if all instances of the calls in the application are protected by mutexes.

/lib/cpp (x86 and PPC)

Some X build utilities depend upon the existence of the file /lib/cpp as they expect to be able to use the C Pre-Processor. LynxOS systems provide the GNU C Pre-Processor, gcc-cpp. As a result, a script file called **cpp** must be created in the directory /lib to provide the required bridge.

The file should contain the lines below with permissions of 755.

x86

```
i386-elf-lynxos/
usr/lib/gcc-lib/i386-elf-lynxos/2.95.3/cpp0 -traditional "$@"
```

PPC

```
/usr/lib/gcc-lib/ppc-elf-lynxos/2.95.3/cpp0 -traditional "$@"
```

The X install script **Install.XM** creates this file at the time of X installation.

CHAPTER 7 *Customizing the Default LynxOS Kernel Configuration*

This chapter explains how to customize the default LynxOS kernel configuration after initial installation.

Reasons for Kernel Customization

As users develop applications in the LynxOS environment, they may need to customize the LynxOS kernel for the following reasons:

- To tune the performance of a LynxOS native development system
- To remove functionality - such as superfluous device drivers - to conserve system resources
- To add functionality - such as networking support or a custom device driver - to support a specific application

By default, the LynxOS kernel on the bootable installation media contains all supported disk and terminal drivers for a given hardware architecture. This allows a single bootable disk/CD-ROM to load LynxOS for initial installation on a wide range of hardware configurations.

To enable LynxOS kernel customization for optimal performance, functionality, and/or size on any given system, during its initial installation, users must make some basic configuration choices. These include the device from which distribution media are to be read, the disk/partition onto which data is loaded, adding networking support, and so on.

To facilitate user's customization decisions, all LynxOS kernel files are located in the `/sys` directory. This centralized location of kernel files allows users to build customized LynxOS kernels quickly and easily.

Customizing for Performance

If LynxOS is being used as a primary development system, it is advantageous to customize the kernel for improved performance.

The following are some of the common features that can be customized for enhanced performance:

- Total number of simultaneously-mountable file systems
- Size of the disk cache
- Maximum number of available processes

Customizing for Size

A user's application environment may have size and memory constraints that make it critical to customize the LynxOS kernel. A kernel's size can be reduced by performing either or both of these tasks:

- Removing unused device drivers
- Tuning performance characteristics for a more minimal environment

For example, LynuxWorks creates a bootable LynxOS kernel image that fits onto a high-density floppy for installation onto certain x86 systems. LynuxWorks customizes the boot kernel so that the kernel image, certain installation utilities (LynuxWorks **`installit`** application), and the RAM disk file system all fit within 1.44 MB.

Customizing for Functionality

The third common scenario involves customizing the functionality of the LynxOS kernel. This includes adding or removing the following components:

- TCP/IP
- NFS
- Simple Kernel Debugger (SKDB)

Users may also need to add their own, custom device drivers for a particular application.

This chapter describes the structure of the LynxOS kernel directory and the process for adding custom drivers to the build environment. For detailed information on developing custom device drivers, see *Writing Device Drivers for LynxOS*.

Overview of the /sys Directory

CAUTION! Read this section before modifying any LynxOS kernel. It contains important information about the `/sys` directory file system that is needed to prevent kernel data corruption and/or loss.

The LynxOS `/sys` directory contains all the scripts, library archives, and platform-specific files that users need to customize and rebuild a LynxOS kernel. The `/sys` directory has the following file structure:

Table 7-1: /sys Directory Contents

File/Directory	Description
`Makefile`	The top-level Makefile
`OBJ_RULES`	Contains the Makefile rules
`bsp.`*`xxx`*	Contains the default Board Support Package (BSP) for the main hardware platforms currently supported by LynxOS. This directory also contains the particular LynxOS kernel parts for any given target board. The kernel image is also modified and linked here.
`cfg`	Contains files that specify each device driver's entry points.
`drivers/bsptype`	Contains the BSP-specific device driver source files.
`include`	Contains the kernel header files
`lib`	Contains the kernel and driver library archives. (See Table 7-2 "sys/lib Kernel Library Files").
`lynx.os`	This is a symbolic link to the `bsp.`*`xxx`* directory.
`romkit`	Contains examples for building bootable kernel images.

Table 7-2: sys/lib Kernel Library Files

Library Name	Description
`libmisc.a`	Miscellaneous support routines
`libnfs_server.a`	NFS server code

Table 7-2: sys/lib Kernel Library Files (Continued)

Library Name	Description
`libsyscalls.a`	System call interface
`libdevices.a`	Common devices
`libdevices.`*`bsptype`*`.a`	BSP-specific devices
`libkernel.a`	Kernel routines
`libnullnfs.a`	Null NFS driver
`libtcpip.a`	TCP/IP code (IPv4 and extras)
`libtcpip6.a`	TCP/IP code (IPv6 and IPSec)
`libdrivers.a`	Common device drivers
`libdrivers.`*`bsptype`*`.a`	BSP-specific device drivers
`libnfs_client.a`	NFS Client code

NOTE: *Debug versions* of all kernel library files also exist in the `/sys/lib` directory, indicated there by a **`*_d.a`** extension.

For information on the contents, functionality, and building a debug version of a kernel library file, see "Creating a Kernel for Debugging" on page 98.

Of all the directories in the `/sys` directory, users are most likely to modify files in the `/bsp.`*`xxx`* directory, as accessed through the `/sys/lynx.os` symbolic link. Users can, however, also modify files in the following `/sys` directories:

- `cfg`
- `devices`
- `dheaders`
- `drivers`

However, these files should only be modified when adding new drivers or when the functionality of a specific driver needs to be changed.

Accessing and Modifying the Main Kernel Directory

The main directory for customizing and rebuilding a LynxOS kernel is accessed through `/sys/lynx.os`. It is a symbolic link to the directory `bsp.`*`xxx`*, where *`xxx`* represents the Board Support Package (BSP) that supports the targeted

hardware (that is, the configuration on which the custom application is to be deployed).

The symbolic link from `lynx.os` to `bsp.xxx` provides the flexibility to add support for target hardware variants, while maintaining a consistent build environment and access to supporting scripts via the `/sys/lynx.os` path.

Overview of /sys/lynx.os

The `/sys/lynx.os` directory contains the following files and subdirectories:

Table 7-3: /sys/lynx.os Directory Contents

File/Directory	Description
`CONFIG.TBL`	The control file used for adding/removing device drivers
`CONFIG.h`	An automatically generated header file
`Makefile`	The target-specific Makefile
`Makefile.common`	The makefile that is used by all LynxOS targets; included in `Makefile`.
`*.o`	Files with this extension are the prebuilt object files.
`info.c`	The device information table
`conf.c`	The configuration file
`uparam.h`	Contains the user-definable kernel parameters from the standard `/usr/include/param.h` parameter set.
`*.h`	Files with this extension are the supporting header files.
`version.c`	Contains LynxOS version information.
`scripts`	Shell functions that are used during a `make install.`*`xxx`*, `make uninstall.`*`xxx`*, or `make all.`*`xxx`*

Rebuilding a Kernel with the make Utility

To rebuild a kernel, change directory to the `/sys/lynx.os` directory, and run the **`make`** utility with a specified Makefile *`rule`* (see Table 7-4 "/sys/lynx.os Makefile Rules"), using the following syntax:

```
# cd /sys/lynx.os
# make rule
```

The following *Makefile rules* are defined in `/sys/lynx.os/Makefile`:

Table 7-4: /sys/lynx.os Makefile Rules

Rule	Description
`all`	Rebuilds **`a.out`** (LynxOS kernel) in the current directory.
`a.out`	Same as `make all`
`clean`	Removes `a.out`, `CONFIG.h`, `sysdevices.h`, `nodetab`, and `timestamp.o`.
`install`	Rebuilds `a.out`, copies `/lynx.os` to `/lynx.os.old`, copies `a.out` to `/lynx.os`, and copies `nodetab` to `/etc/nodetab`.
`install.`*`xxx`*	Adds the *`xxx`* functionality (`tcpip`, `nfs`, `skdb`, or `all`) to `a.out`, then installs it. For more information, see the **`install`** man page.
`uninstall.`*`xxx`*	Removes the *`xxx`* functionality (`tcpip`, `tcpip6`, `nfs`, `skdb`, or `all`) from `a.out` then installs it.
`all.`*`xxx`*	Adds the *`xxx`* functionality (`tcpip`, `tcpip6`, `nfs`, `skdb`, or `all`) to `a.out`
`SYS_DEBUG=true`	Can build a debug kernel with hardware watchpoint support and code test.

NOTE: The implication of the `install` rule in the kernel Makefile is as follows: The process of "installing" a kernel involves copying the kernel to the root directory as `/lynx.os` and copying the device `nodetab` file to `/etc/nodetab`. The "installed" kernel, `/lynx.os`, is usually the default bootable kernel at system startup. Old kernel files are copied to `*.old`.

Incorporating Changes Made in /sys/devices into the Kernel

To incorporate changes made in `/sys/devices` into the kernel, perform the following steps:

1. Change to the `/sys/devices` directory.

```
# cd /sys/devices
```

2. Add or update the appropriate device file to the `/sys/devices` directory.
3. Rebuild the kernel. For example:

```
# make install
```

Functionality Scripts

Users can also use the scripts in `/usr/bin` to install or remove the following major functional modules:

- TCP/IP, NFS, PPP, Samba, SNMP
- Simple Kernel Debugger (SKDB) support

For example, to add support for NFS, enter the following command:

```
# /usr/bin/Install.nfs
```

To remove SKDB support from the kernel, enter the following command:

```
# /usr/bin/Uninstall.skdb
```

To add support for TCPIP (IPv4 and extras), enter the following command:

```
# /usr/bin/Install.tcpip
```

Please note that if `Install.tcpip6` is entered in place of `Install.tcpip`, you will add support for IPv6 and IPSec to the kernel.

Adding/Removing Device Drivers with CONFIG.TBL

The `CONFIG.TBL` file controls the device drivers that are inserted into the kernel at build time. Each driver has a one line entry in this file. Inserting the comment character (#) in front of the line deactivates the given device driver. For example, the code fragment below shows two lines from `/sys/lynx.os/CONFIG.TBL` on a LynxOS x86 system:

```
# Secondary ide controller
I:ide.cfg
```

This fragment indicates that the LynxOS kernel built with this `CONFIG.TBL` contains a device driver for the secondary IDE controller. To remove this device driver, modify the lines in `CONFIG.TBL` as shown below:

```
# Secondary ide controller
#I:ide.cfg
```

Similarly to enable support for TCPIP6, the following lines from `CONFIG.TBL` must be uncommented, as follows:

```
I:hbtcpip.cfg
n:hbtcpipv6:@hbtcpip0::
```

To reflect this change, the kernel must be rebuilt.

CAUTION! When building a new kernel after changing `CONFIG.TBL`, the device nodes for the new kernel may not match the previous nodes. The device nodes should be built after making the new kernel; for more information, see the **`mknod`** man page.

Whenever `CONFIG.TBL` is changed to build a new kernel, the device nodes for the new kernel must also be rebuilt. Since the major and minor device numbers may change when editing `CONFIG.TBL`, users may also have to modify the root device settings to boot LynxOS.

Here are some guidelines for managing changes to `CONFIG.TBL`:

- When adding or removing a disk driver (usually specified at the beginning of `CONFIG.TBL`), the user must determine the new major and minor number of the root device to boot the new kernel; See “Removing Unused Device Drivers” on page 101.
- Because functionality modules (similar to TCP/IP drivers) are specified *after* the disk drivers, the major and minor number of the root device does not need to be changed to boot the new kernel.
- When creating a custom device driver, users should add it to the bottom of `CONFIG.TBL` so that the device node information for the preceding devices stays the same.

Making a Personal Kernel Build Directory

When working on team projects, developers may need to maintain individual kernel build environments. The `/sys/lynx.os` directory was designed with this scenario in mind.

The `lynx.os` directory (`bsp.`*`xxx`*) is position-independent. That is, it can be copied to any location on the file system and then used to build a *local* kernel (`a.out`) from the copy.

The Makefile in the `lynx.os` directory uses the environment variable **`$ENV_PREFIX`** to find the library archives and object rules, but it is otherwise

ignorant of its location on the disk. `$ENV_PREFIX` defaults to the root directory (`/`) on native development systems. On cross development systems, the shell script `SETUP.shell` defines `$ENV_PREFIX` when executed: (See "Setting the Cross Development Environment" on page 12 for more information).

```
# cp -r /sys/lynx.os /tmp/my_lynx.os
# cd /tmp/my_lynx.os
# touch CONFIG.TBL
# make a.out
```

In this way, multiple developers can have multiple kernel build directories on a given system, each with its own unique `CONFIG.TBL`, `nodetab`, and `uparam.h` files.

CAUTION! When building native LynxOS kernels, do not invoke the `make install` command. This option causes the resulting local kernel to overwrite `/lynx.os` and for the new node table to overwrite `/etc/nodetab`.

Customizing from a Cross Development Host

The LynxOS distribution for cross development environments provides users with two shell scripts, which define several key environment variables such as **`PATH`** and **`ENV_PREFIX`**. These shell scripts, **`SETUP.bash`** and **`SETUP.csh`**, are located in `/usr/lynx/xx/platform`, where *`platform`* is `x86` or `ppc`. One or the other of these scripts, depending on the shell requirements of the developer, must always be executed before developing applications with LynxOS tools.

The scripts are invoked using the following syntax (in the examples below, for LynxOS Release *`xxx`*, loaded onto an x86 development host):

- For the `.csh` shell:

```
$ cd <install_dir>/usr/lynx/xxx/x86
$ source SETUP.csh
```

- For the `bash` shell:

```
$ cd <install_dir>/usr/lynx/xxx/x86
$ . SETUP.bash
```

Where `<install_dir>` is the LynxOS installation directory.

With the exception of having to first invoke a `SETUP` script on cross development systems, *there is no discernible difference in the development process of LynxOS kernels from cross or native machines*. LynxOS uses the same Makefiles and directory structure on cross development systems as on native systems.

The design of the `/sys` directory and Makefile system provides this functionality. The following example demonstrates the similarities in developing kernels on cross development and native LynxOS systems.

Adding TCP/IP to a LynxOS Kernel

- On a Cross Development System

  In the following commands, TCP/IP IPv4 is added to an x86 cross development system:

  ```
  $ cd /usr/lynx/x86
  $ . SETUP.bash
  ```

  The following commands then set up the `PATH` variable that points to the cross development tools, and sets `$ENV_PREFIX` to point to the installed LynxOS directory tree:

  ```
  $ cd $ENV_PREFIX/sys/lynx.os
  $ make install.tcpip
  ```

  In this example, the LynxOS kernel (`a.out`) and the devices `nodetab` are copied to `ENV_PREFIX`.

  Additionally, the `/usr/bin/Install.tcpip` script can be used to install TCP/IP onto a system.

- On a Native Development System

  In the following commands, TCP/IP is added to a native development system:

  ```
  $ cd /sys/lynx.os
  $ make install.tcpip
  ```

  In this example, the LynxOS kernel (`a.out`) and the devices `nodetab` are copied to `$ENV_PREFIX` on a native LynxOS system.

  Additionally, the `/usr/bin/Install.tcpip` script can be used to install IPv4 onto a system.

`install.tcpip6` will install IPv6, IPSec onto a system. Consult your LynuxWorks sales representative if you would like IPv6/IPSec features.

NOTE: This procedure applies to IPv4 only.

Customizing a Kernel for Performance

The parameters defined in `/sys/lynx.os/uparam.h` tell the kernel how much space (dynamic memory) to set aside for various data structures at run-time. Users can modify this file to increase (or decrease) the default values.

Configurable Parameters in /sys/lynx.os/uparam.h

The table entitled "Default Values for /sys/lynx.os/uparam.h" on page 96 lists some of the more important configurable parameters in the binary code written for a standard development LynxOS system installation in the `/sys/lynx.os/uparam.h` file. Other configurable parameters are also contained in this file, and can be found by searching for the character string **`#define`**.

The `/sys/lynx.os/uparam.h` file can be viewed using the **`more`** command or in a text editor such as vi.

Modification to the values of these parameters not only influences system performance, but also increases or decreases the dynamic size of the kernel.

NOTE: Before making any changes to `/sys/lynx.os/uparam.h`, users need to read the comments in the files for any parameter that has been changed. Some parameters depend on others. For example, the value of `/sys/lynx.os/uparam.h/NFILES` must be greater than or equal to the sum of all the file parameters, including message queue, shared memory, and semaphore special files. Otherwise, there is no room for these files and the system does not allow them to be created or stored.

Parameter Default Values in /sys/lynx.os/uparam.h

The configurable parameters and their default values in the `/sys/lynx.os/uparam.h` file are listed in the table below:

Table 7-5: Default Values for /sys/lynx.os/uparam.h

Parameter Name	What the Parameter Defines	Default Value
CACHEBLKS	Disk cache blocks	1024
MAX_FAST_SEMS	Fast semaphores	512
MAXSYMLINKS	Symbolic links in a path	8
NBDEVS	Dynamically-loaded block devices	10
NCDEVS	Dynamically-loaded character devices	20
NDRIVS	Dynamically-loaded drivers	15
NFILES	Open files in system	256
NMOUNTS	Mounted file systems	10
NINODES	In-cache inodes	256
NPIPES	Pipes	50
NPROC	Processes	50
NRLOCKS	File locks	140
NSHLIB	Shared libraries	10
NSOCKETS	Sockets	100
NSTASKS	Kernel threads	20
NTHREADS	User threads	50
NUMIOINTS	Registrable interrupt vectors via `iointset()`; combined number of interrupt vectors derived by combining number of hardware vectors with software multiplexes	256
NVSEMS	Number of SystemV headers	20
NVSHMS	Number of SystemV-shared memory segments	20
QUANTUM	Clock ticks until preemption	25
SMEMS	LynxOS IPC named shared memory segments	10

Table 7-5: Default Values for /sys/lynx.os/uparam.h (Continued)

Parameter Name	What the Parameter Defines	Default Value
USEMS	LynxOS IPC user semaphores	20
USR_NFDS	Open files per process	40
USYNCH	LynxOS IPC pages user synchronization objects	50

Additional Configurable Parameters for Detecting Fatal Errors on an MCP750 Board

Users can enable fatal error detection for LynxOS systems installed in processors mounted on an MCP750 board by adding the following line to the `uparam.h` file, where *xxx* is the special device file for a flash memory driver:

```
#define CRASH_DEVNAME xxx
```

NOTE: Not setting `CRASH_DEVNAME` on MCP750 board systems automatically sets this feature to disabled.

Increasing Maximum Processes

This example illustrates how to customize a LynxOS kernel to handle a greater number of processes. It assumes that the default bootable kernel, `/lynx.os`, is being modified in order to:

- Modify `/sys/lynx.os/uparam.h`
- Make a new kernel
- Reboot the system

Perform the following steps:

1. Make a backup copy of `uparam.h` (to revert to the kernel's original settings) using the following command:

   ```
   $ cd /sys/lynx.os
   $ cp uparam.h uparam.h.old
   ```

2. Use a text editor, such as vi, to edit `uparam.h`.
3. Find the parameter `NPROC`, and increase its number, as follows:

```
#define NPROC xxx /* max number of\ processes */
```

NOTE: The `NPROC` parameter is being increased to increase the maximum number of processes. Here, *xxx* represents the `NPROC` number.

4. To activate the changes, rebuild the kernel and reboot the system using the following commands:

```
$ make install
$ reboot -a
```

Creating a Kernel for Debugging

Users can create debug versions of the LynxOS kernel for use with SKDB and GDB. The debug version of the LynxOS kernel includes support for:

- Hardware Watchpoints
- Code Test

For more information on these functions, see the *Total/db User's Guide.*

To create a debug version of the LynxOS kernel, use the following when invoking make:

```
# make all SYS_DEBUG=true
```

NOTE: The debug version of the LynxOS kernel should not be used for deployment. Rebuilding the kernel without the debug information results in improved performance.

Changing Kernel Size

Usually, users change the size of a kernel to decrease the amount of storage space it needs and/or the amount of memory it needs for execution. This section shows the components of the kernel that affect its size, both in terms of memory and storage.

There are four basic components that users can modify to change the size of the LynxOS kernel:

- System parameters in `/sys/lynx.os/uparam.h`

- Functionality modules (TCP/IP, NFS, and so on)
- Device drivers
- Symbol table information

Of these, `uparam.h` is the only component that is *dynamic* - changes to `uparam.h` affect the kernel size in memory. The other components mainly affect the *static* size of the kernel - the amount of space it takes to store on a device. However, if users reduce the static size of the kernel, they usually reduce some of its dynamic size as well. The only exception is the symbol table information - removing this only affects the static size of the kernel.

After determining the components to modify, users must make a new kernel to see the size changes in effect. This process is similar to tuning a kernel for performance.

Determining the Kernel Size

The following subsections detail how to experiment with changing a kernel and determine its size.

Determining Kernel Disk Space Usage

An easy way to determine the how much total disk space the kernel is using, is using the **`ls -l`** command, as shown below:

```
$ ls -l /lynx.os
-rwxr-xr-x 1 root 877060 Oct 29 12:01 /lynx.os
```

The kernel in this example uses 877060 bytes, or about 877 KB of space.

The **`size`** command allows users to look more closely at the sizes of the **`text`**, **`data`**, and **`bss`** segments of the kernel, as illustrated by the command and output below:

```
$ size /lynx.os

text    data   bss    dec     hex    filename
688096  45056  142300  875452  d5bbc  /a.out
```

This utility reports the size (in bytes) of each segment and the sum of the segments. The total size of the file **`a.out`**, in this example, is 875452 bytes (or `0xd5bbc`).

In addition to the `text` and `data` segments, the kernel contains header and symbol table information that take up the additional 1608 bytes, as shown in the command output above.

Determining Kernel Memory Usage

Figuring out the memory usage of the kernel is more difficult than determining its static size. Users can use the **ps** command to look at a currently running LynxOS kernel.

The following example shows how to use **ps** to look at a currently running LynxOS kernel. On a LynxOS system, enter the following command:

```
$ ps -axonT
```

This command displays all currently executing processes, including threads and unattached processes, and reports the total free memory (both physical and virtual) in kilobytes (KB) at the end of the listing:

```
pid tid ppid pgrp pri text stk data time  dev  user S name
0   0   0    0    0   0    0   0    25:49      root R nullpr
1   7   1    1    16  20   8   12   2.44       root W /init
20  12  1    20   18  12   8   8    0.08  atc0 root W /bin/syncer
23  48  26   23   17  56   8   36   0.10       root R /net/rlogind
26  17  1    26   17  52   8   40   0.01       root W /net/inetd
28  18  1    28   17  40   8   52   0.00       root W /net/unfsio
31  19  1    31   17  44   12  32   0.03       root W /net/portmap
33  20  1    33   17  56   8   52   0.00       root W /net/mountd
35  21  1    35   17  44   8   40   0.00       root W /net/nfsd
37  22  1    37   17  64   8   44   0.00       root W /net/rpc.statd
39  23  1    39   17  60   8   60   0.00       root W /net/rpc.lockd.svc
41  24  1    41   17  60   8   68   0.02       root W /net/rpc.lockd.clnt
43  11  1    43   17  72   8   40   0.00       root W /bin/lpd
45  14  1    45   17  204  28  128  0.26  atc0 root W /bin/bash
46  27  1    46   17  204* 28  128  0.27  atc1 root W /bin/bash
47  30  1    47   17  204* 28  136  0.74  atc2 root W /bin/bash
48  33  1    48   17  204* 28  124  0.19  atc3 root W /bin/bash
63  45  46   63   17  340  8   72   0.02  atc1 root S /bin/vi
78  41  23   78   17  204* 28  124  0.08  ttyp0root W /bin/bash
119 54  45   119  17  40   8   36   0.08  atc0 root W /bin/rlogin
121 55  78   121  17  40   12  40   0.07  ttyp0root C /bin/ps
137 46  48   137  17  340* 12  72   0.04  atc3 root S /bin/vi
147 40  119  119  17  40*  8   36   0.06  atc0 root W /bin/rlogin
8352K/0K free physical/virtual, 2872K used (in this display)
```

To calculate the size of the kernel:

1. Add the amount of free physical memory (8352 KB) to the amount used (2872 KB): 8352 + 2872 = 11224 KB.
2. Subtract this number from the total amount of memory (RAM) on the LynxOS system.

   The machine in this example has 16 MB of memory, which translates to 16384 KB: 16384 – 11224 = 5160 KB.

Removing Unused Device Drivers

One way to change to the size of a kernel is to remove unused device drivers, as described in the example below:

This example is based on a LynxOS x86 system that has only one Adaptec 2940 SCSI adapter to control its disk and CD-ROM drives. It does not need device entries (space used by the kernel) for other LynxOS supported SCSI device drivers.

Before Beginning

1. Back up the `/sys/lynx.os` files and directories to `/tmp/test` by entering the following commands:

   ```
   $ cp -r /sys/lynx.os /tmp/test
   ```

2. Change to the `/tmp/test` directory.

   ```
   $ cd /tmp/test
   ```

3. Record the following information about the kernel:

- The `/lynx.os` file's current size (to be used later in validating that changes have taken effect)
- The root file system's device name
- The root file system device's major and minor numbers
  - Use the **`ls -l`** or **`size`** command to determine the `/lynx.os` files's current size:

    ```
    $ ls -l /lynx.os
    -rwxr-xr-x 1 root 698924 Oct 29 12:01 /lynx.os
    ```

  - Use the **`df`** command to determine the root file system's device name. In this example, the root file system device name is `/dev/sd2940.0a`.

    ```
    $ df /
    /dev/sd2940.0a   2048  127992  36979   91013  28%
    ```

  - Use the **`ls -l`** command to determine the root file system device's major and minor numbers. In this example, the major number is `4` and the minor number is `16`.

    ```
    $ ls -l /dev/sd2940.0a
    brw------- 1 root 4,16 Oct 29 02:33 /dev/sd2940.0a
    ```

Modifying CONFIG.TBL

Use a text editor (such as vi) to modify `/tmp/test/CONFIG.TBL` as shown below to comment out unnecessary drivers by adding a pound character (#) to the beginning of the line:

```
# Supported LynxOS Adaptec Scsi drivers
#       for SCSI Manager-based Adaptec 154x driver
#I:sim1542.cfg
# NCR 810 SIM (SCSI Interface Module) driver.
#This SCSI driver uses SCSI Manager.
#I: simncr.cfg
#       for SCSI Manager-based Adaptec 294x driver
I:sim2940.cfg

#I:scsi810.cfg
#       for high-level-low-level based ncr 810 driver
```

Building the Newly Customized Kernel

Use the following procedure to build a new kernel:

1. Change the working directory to `/tmp/test`.

   ```
   $ /tmp/test
   ```

2. Build and install a new LynxOS kernel by entering the following command:

   ```
   $ make install
   ```

   After building the new kernel, the installation script generates the file `/tmp/test/a.out` and copies it to `/lynx.os`. Additionally, the `install` script first creates, then copies a new `nodetab` file from `/tmp/test/nodetab` to `/etc/nodetab`, and then generates the file `/etc/nodetab.old`.

NOTE: The older versions of generated files are automatically renamed with an `.old` extension.

---

3. Verify that the new kernel has been created by entering the following command:

   ```
   $ ls -l /lynx.os*
   -rwxr-xr-x 1 root 661669 Oct 29 12:01 /lynx.os
   ```

```
-rwxr-xr-x 1 root 698924 Oct 29 12:01 /lynx.os.old
```

In this example, the size of the new kernel is smaller than the previous kernel by about 36 KB, verifying that the new kernel has accepted the configuration changes made earlier.

Making New Device Nodes

Use the following procedure to create new device nodes.

1. Make a copy of the old devices directory:

   ```
   $ mv /dev /dev.old
   $ mkdir /dev
   ```

2. Build new device nodes that correspond with the new `/etc/nodetab` file by entering the following commands:

   ```
   $ cd /dev
   $ mknod -a /etc/nodetab
   ```

Loading the New Kernel

Users must know the major and minor number of the new kernel's root file system before rebooting the machine (notice that the major number has changed to `2` in the output below).

1. Enter the following command to get the new kernel number:

   ```
   $ ls -l /dev/sd2940.0a
   brw------- 1 root 2,16 Oct 29 02:33 /dev/sd2940.0a
   ```

CAUTION! If either a wrong major or minor number is specified as the root device, the error message `Main file system failed to install (no dev)` may be displayed, or the kernel may crash.

2. Reboot the system by entering the following command:

   ```
   $ reboot -a
   ```

3. At the **`preboot`** command prompt (the bootstrap kernel loader program), specify the new major and minor number of the root device:

   ```
   $ Command? R 2 16
   ```

```
$ Setting root device to 2 16
$ Command? b
```

See the `preboot` man page for a description of the commands available on the given hardware platform.

Updating the Default Root Device

Now that the new kernel is up and running, users can update the default root device with the **`makeboot`** command:

```
$ makeboot preboot
```

This command allows users to directly boot LynxOS without using the **`R`** command in `preboot`.

Adding Functionality to a Kernel

Many LynxOS users need to develop device drivers that are specific to their target hardware. LynxOS supports this development with the `/sys/lynx.os/Makefile` file. In this Makefile, users can see several Makefile environment variables that they can modify to explicitly specify custom device drivers, patches, or other objects to include in the kernel.

Adding a Custom Device Driver

LynuxWorks recommends the following approach for adding a custom driver to a LynxOS kernel. Substitute actual file and directory names where appropriate in the example below.

1. Create a custom driver directory in `/sys/drivers`:

   ```
   $ mkdir /sys/drivers/my_driver
   ```

2. Use an existing device driver Makefile as a template by entering the following commands:

   ```
   $ cp /sys/drivers/mem/Makefile
   $ /sys/drivers/my_driver
   ```

3. Modify the Makefile so that during a kernel build, the driver is compiled into a unique archive:

```
$ FILES = file1.x file2.x file3.x
$ LIBRARY=my_driver
```

These lines instruct the Makefile to compile `file*.c` files, and to insert the compiled `.o` files in `/sys/lib/my_driver.a`. Users can look at the `OBJ_RULES` file in the `/sys/` directory to determine the substitutions that are appropriate for their compilation of the `.c` files. For example, `.x` compiles an optimized ANSI `.c` file, while `.u.x` does not optimize the `.c` file.

4. Modify the Makefile in `/sys/lynx.os` so that the archive for the device driver is linked to the kernel.

```
$ CUSTOMER_DRIVERS=$(LIB)/my_driver.a
```

5. Add the configuration file `my_driver.cfg` for the driver in `/sys/cfg`; see *Writing Device Drivers for LynxOS* for more information about device configuration files.
6. Edit the `/sys/lynx.os/CONFIG.TBL` so that the new driver is represented:

```
$ I:my_driver.cfg # for instance
```

NOTE: New drivers should always be added to the end of `CONFIG.TBL` so that the major and minor numbers for existing device drivers remain the same:

7. Build the LynxOS kernel with the new device driver by entering the following commands:

```
$ cd /sys/lynx.os
$ make install
```

Configurable Tick Timer

Users can configure the number of ticks per second for real-time clocks. The define `TICKSPERSEC` in the `/usr/include/conf.h` file defaults to 100 ticks per second.

LynuxWorks recommends the following minimum and maximum ticks per second:

Table 7-6: Recommended Ticks Per Second value of TICKSPERSEC

Minimum ticks per second	Maximum ticks per second
20 (50ms between ticks)	500 (2ms between ticks)

Configuring Core Files

LynxOS provides the ability to configure the information saved in core files. This allows users to control the size of the core file. This feature can be useful in systems with limited memory or disk resources by storing only the data essential for debugging the application at hand.

Configurable Options

The following options can be set when installing the configurable core file support:

- Data section (global and static initialized data)
- BSS section (global and static uninitialized data)
- Program heap section (memory allocated by the program using the `malloc()` or `sbrk()` interfaces)
- Shared memory (the memory segments inherited from the shared libraries used by the program and the memory obtained using the `shmget(2)` interface). The data and BSS sections of the shared libraries are considered a part of shared memory.

NOTE: Turning off any of the options described above may affect the postmortem debugging of the program with the LynxOS GDB debugger. For a detailed description, see *The Total/DB User's Guide*.

Additionally, users can define the name and location of the core file.

- The POSIX standard defines the name of the core file as "`core`". Users can select an non-POSIX alternative file name. The core file name appends the current seconds value of the core dump event, as well as the PID of the dumping application. For example:

  `core.`*`seconds`*`.`*`pid`*

- The LynxOS kernel can be configured to save the core file in a specific location, rather than in the application's current directory. Both relative and absolute paths are supported.

Installing Configurable Core File Capability

Install the configurable core file capability by running either of the following commands:

- Run the `Install.core` script:

  ```
  # /usr/bin/Install.core
  ```

  or

- Use `make install.core` to enable the configurable core file:

  ```
  # cd /sys/lynx.os
  # make install.core
  ```

Either option starts the same configuration script, allowing the user to update the user defines in `uparam.h`.

The following figure details a sample configuration dialog session:

```
bash# cd /sys/lynx.os
bash# make install.core
/bin/make PARSE=Done COMPONENTS="core" install.all

Configuring the Core File Feature ...

Dump the data section [y,n] (y): Enter
Dump the bss section [y,n] (y): Enter
Dump the heap section [y,n] (y): Enter
Dump the shared memory section [y,n] (y): n
Enable Non-POSIX core file naming convention [y,n] (n): y
Please enter the location where the core should be placed [.]: /tmp/altcore
```

Figure 7-1: Sample Configuration Dialog

NOTE: The following combination of the core configuration settings is considered illegal:

- Data section dump is on
- BSS section dump is off
- Heap section dump is on.

With these configuration settings, the data, BSS, and heap sections dumps cannot be saved to the core file in one section as defined by the format of the core. If these configurations are defined, the BSS section will be dumped into the core file.

Restoring the Defaults Settings

To restore the default core file settings, the user must run one of the following commands:

- Run the `Uninstall.core` script located in the `/usr/bin` directory.

  ```
  # /usr/bin/Uninstall.core
  ```

  or

- Issue the `make uninstall.core` command from the `/sys/lynx.os` directory.

  ```
  # cd /sys/lynx.os
  # make uninstall.core
  ```

User Definitions in uparam.h

The configurable core file includes a number of user defines in the LynxOS `uparam.h` header file. The following table includes the user definitions and default values:

Table 7-7: User Defines in uparam.h

User Defines	Default Value
CORE_WRITE_DATA	1
CORE_WRITE_BSS	1
CORE_WRITE_HEAP	1

Table 7-7: User Defines in uparam.h (Continued)

User Defines	Default Value
CORE_WRITE_SHARED	1
CORE_NONPOSIX_FILENAME	0
CORE_ALTERNATE_LOCATION	NULL

Default Configuration of the Core File

The default settings of the configurable core file in `uparam.h` are as follows:

- Program data section dump is on
- Program BSS section dump is on
- Program heap section dump is on
- Program shared memory dump is on
- Non-POSIX core file name is off
- Core file location is the current working directory

These default settings create a conventional core file, effectively disabling any configuration options defined by the configurable core file. Core files created when the default settings are defined in `uparam.h` are backward compatible with non-configurable core files.

CHAPTER 8 *Creating Kernel Downloadable Images (KDIs)*

LynxOS supports creating Kernel Downloadable Images (KDIs) with the **`mkimage`** utility. KDIs combine the LynxOS kernel and its associated applications into bootable images designed for easy downloading onto development targets.

Overview

Users can create application-ready embedded systems that include the LynxOS kernel and customized application into bootable images, which can then be implemented in one of the following ways:

- Burned into Flash
- Put onto distribution media (Floppy or CD-ROM)
- Booted over a network
- Maintained on hard drives

A LynxOS kernel image is a special boot file that always contains LynxOS, but may also contain a RAM-based file system (called a RAM disk) with minimal files for operating system functions and applications.

mkimage - the LynxOS KDI Creation Utility

The `mkimage` utility creates a single file - a bootable kernel downloadable image (KDI) - with some or all of the following LynxOS components:

- The LynxOS kernel
- A memory image of a RAM disk with a specified initial set of files
- The text segments of applications loaded at initialization time

The `mkimage` utility creates the image with attributes that users set in a specification file. By defining the fields in the specification file, users can create customized images for their specific development targets. For more information on specification files, see “Creating a Specification File” on page 116.

NOTE: See the `mkimage.spec` man page for an explanation of specification files.

The mkimage Syntax and Features

The syntax for `mkimage` is as follows:

```
# mkimage spec_file output_file
```

NOTE: Additional options are available in the `mkimage` man page.

- The `mkimage` utility lets users create bootable kernel downloadable images in any directory, with any number of specification files.
- Provided that all of the specification files are accessible, users do not need root privileges to run `mkimage`.
- The `mkimage` utility runs on both native LynxOS systems or cross development hosts.

LynxOS Kernel

Users can specify that `mkimage` should use the current LynxOS kernel in `/lynx.os` when building the KDI. However, users are most likely to create a custom kernel for use in their specific image. For example, users may choose to remove any unused device drivers from the kernel to reduce the size of the kernel, and as a result, of the final image.

(See Chapter 7, “Customizing the Default LynxOS Kernel Configuration” for more information on creating custom kernels.)

To save space, `mkimage` does not expand the `bss` section of the kernel in the image. The `bss` section is expanded and initialized to zeros by kernel code. The starting execution address is always 32 (`0x20`) bytes after the start of the image in memory.

Embedded File Systems

Embedded file system images can be one of two types:

- An image that may be used as the root file system contained in, or referenced by, the kernel image.
- A stand-alone file system image that may be mounted as an additional file system after LynxOS is up and running. The `mkimage` utility prepares both standalone, and kernel root file systems.

Embedded Root File Systems

Embedded root file systems can be one of the following three types:

- RAM-Based File System

  The `mkimage` utility may be used to create a memory-based root file system. LynxOS accesses this file system using its RAM disk driver. The memory-based file system may physically reside in RAM, ROM or flash memory, depending on how the image is used and created for the target machine. Depending on the set of utilities put into the image, the image may function as a minimal application target or as a complete LynxOS development system.

- Disk-Based File System

  If the major and minor numbers of a disk drive attached to a target system are specified, that drive may be used as the root file system for the kernel downloadable image (KDI). A disk-based root file system behaves identically to a kernel root file system booted from disk.

- No File System

  For kernels appropriately generated, the image does not contain or reference a file system.

Embedded Stand-Alone File System Images

An embedded standalone file system image is memory-based and may be accessed through the LynxOS RAM disk driver. The file system image may reside in any type of physical memory but is most likely to be in ROM or flash memory. No provision for moving the embedded file system image to RAM is provided by the

LynxOS kernel. Application text segments may or may not be resident as required (see “Resident Text Segments” below).

Resident Text Segments

Applications may have resident (execute in place - XIP) text images in an embedded memory-based file system. In a normal file system, the text segment of an application is copied from the file system to RAM for execution. In a memory-based file system, the text segment is copied from the memory-resident file system to memory. Resident text applications do not require this copying process, which in turn, reduces run-time RAM memory requirements. The considerations that should be taken into account when deciding to use resident text are as follows:

- The type of memory the resident text is in
- Required execution speed
- RAM constraints
- Type of embedded file system

NOTE: Be aware that specifying `text=ram` for a root file system image that normally resides in some type of ROM, copies all of the application’s resident text in the image to RAM during kernel initialization.

When the execution speed of the application is most important, the text of the image should reside in RAM, because, in general, any type of ROM is slower than RAM. There are several ways to accomplish this, depending on how the KDI or File System Image (FSI) is specified, and the type of memory the image resides in.

When the application is in the root FSI, and the KDI is in RAM, resident text should be used to ensure optimal execution speed and RAM conservation.

When the application is in the root FSI and the KDI is in some sort of ROM, the text may or may not need to be resident. When the text is specified as resident and `text=ram` is also specified, the resident text of the application (and all other resident text applications) is copied to RAM by the kernel during kernel initialization. When the text is not specified resident, the kernel loads the applications text to RAM from the file system at execution, identically to the application text residing on a disk drive. Since there is no provision to move resident text segments to RAM in a standalone FSI, a non-resident text segment is

the only way to have the application run out of RAM from a standalone FSI residing in some sort of ROM.

When the root or standalone FSI resides in ROM and execution speed is a priority, leaving the application text segments in the non-resident files system conserves the greatest amount of memory. This causes the kernel to load the text segments on demand into RAM from the FSI. Alternately, when the root FSI resides in RAM, using resident text for applications conserves the greatest amount of memory.

Creating a KDI Image

Procedure Overview

The same basic steps are used to create images for the entire range of boot applications, for network booting to ROM booting:

- Configure a LynxOS kernel with the desired functionality.
- Create a specification file for `mkimage` that defines the LynxOS kernel to use and any applications to include.
- Run `mkimage`.
- Test the image.
- Put the image into the target environment.

For information on network booting the image, please read “Network Booting Diskless Clients with LynxOS” in the *LynxOS Networking Guide*.

The following steps detail creating a Kernel Downloadable Image (KDI). These steps are also explained in the subsections that follow:

1. Create a copy of `/sys/lynx.os` by entering the following commands:

   ```
   $ cp -r /sys/lynx.os /tmp/test.os
   $ cd /tmp/test.os
   ```

2. Enable the RAM disk driver in `CONFIG.TBL`.
3. Make any other modifications to files such as `uparam.h` and `Makefile`.
4. Create a new kernel with `make a.out`.
5. Create a specification file with desired attributes.

6. Run `mkimage`.

For information on creating a remotely bootable kernel (including sample scripts), see "Network Booting Diskless Clients with LynxOS" in the *LynxOS Networking Guide*.

NOTE: When developing application on a LynxOS cross development host, it is necessary to use `$ENV_PREFIX/pathname` where absolute paths are shown in this chapter.

Enabling the RAM Disk Driver

To enable the RAM disk driver, users must uncomment (by removing the `#` sign) the line `I:ramdisk.cfg` in the `CONFIG.TBL` file.

Modifying Kernel Parameters

One way to reduce the size of a kernel is to remove all superfluous drivers; see Chapter 7, "Customizing the Default LynxOS Kernel Configuration" for examples.

Creating a Specification File

The example specification file in the `mkimage.spec` man page can be used as a template for creating a specification file.

Some important attributes of `mkimage.spec` are as follows:

Table 8-1: mkimage.spec Attributes

Attribute	Description
`target=target=[x86|ppc]`	The target system
`osstrip=[local|all|none]`	Causes local symbol definitions to be stripped from the kernel text file.
`ostext=[ram|rom]`	Designates where the kernel resides in the running system.
`kernel=path`	The path of the LynxOS kernel to be used in the image; This can be `/lynx.os`, but most likely is a customized kernel that the user has built elsewhere on the system.
`nodetab=path`	The device node table corresponding to the kernel

Table 8-1: mkimage.spec Attributes (Continued)

Attribute	Description
`flags=`	Designates the boot flags.
`free=`	Designates the number of free blocks.
`inodes=`	Designates the number of free inodes.
`root=[ram|rom]`	Specifies that the root file system is either resident in RAM, ROM, mounted from the device, or that there is no file system.
`strip=[true|false]`	Designates whether or not the application files are to have symbols stripped.
`text=[ram|rom]`	Designates that the resident application programs will be in ROM, or moved to RAM.
`resident=[true|false]`	Designates whether or not the application programs are to be resident in memory.
`directory=`	A directory on the target file system
`file=`	A file on the target file system
`source=`	Designates a fully qualified path name to a source file to be copied into the target file system as the file specified in the `file=`.
`owner=`	Designates the numeric owner ID of the target file.
`group=`	Designates the numeric group ID of the target file.
`mode=`	Designates the file mode for the target `filesymlink`.
`symlink`	Designates a symbolic link of *`pathname1`* to *`pathname2`*.

Testing Kernel Images

Users can use the LynxOS `preboot` utility to directly test the kernel images made with `mkimage` (see the `preboot` man page for the specific commands used on different platforms). For example, to test the kernel image `/tmp/test/images/image1` created on a LynxOS x86 machine, perform the following steps:

1. Reboot the machine by entering the following command:

   ```
   $ reboot -a
   ```

2. At the `preboot` prompt (`Command?`), enter the following command:

```
Command? b /tmp/test/images/image1
```

Booting KDIs

The following sections provide details on:

- Booting images over the network (remote booting)
- Booting from ROM

Booting Images over a Network

To network boot LynxOS (also called to *netboot*), the image file is stored on a remote server and copied over the network into the RAM of a remote node by the firmware-resident TFTP boot code. Generally, the node is diskless.

Because everything is in RAM, the boot code gives control to the loaded operating system, which in turn, configures the attached RAM disk as its `root` file system. In this case, because all the executables are in RAM, it is useful if they are executed directly without being copied again into RAM. Thus, the `TEXT` segments of the executables are loaded as part of system initialization at boot time, rather than on demand from the root file system.

In the remote boot configuration, the RAM memory map is sequential, as shown in the figure below:

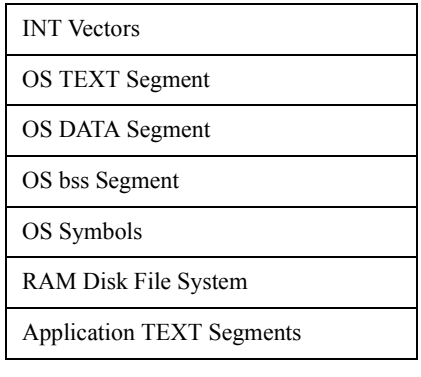

Figure 8-1: Example RAM Memory Map

Booting Images from ROM

The firmware-resident ROM boot code always passes control to the ROM-based kernels. At this point, depending on the options defined in the specification file, all or portions of the image are moved to RAM. LynxOS always moves the `DATA` and `bss` sections to RAM.

LynxOS sets up the memory map such that some portions are mapped into ROM and other portions are mapped into RAM. This is required in order to fully utilize the contents of ROM, without copying them into RAM. The previous figure shows an example of this mapping scheme. The LynxOS kernel text segments, root file system, and resident text segments are all in ROM, as shown in the figure below.

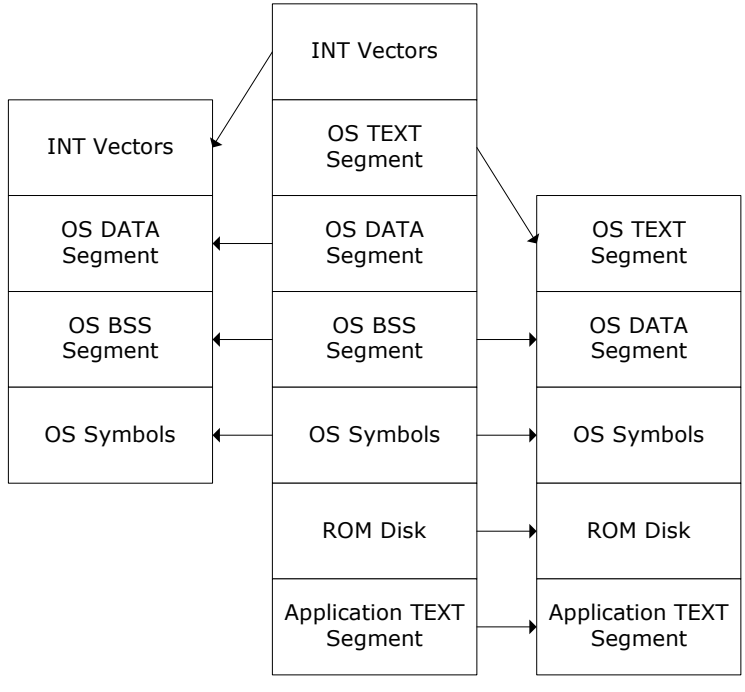

Figure 8-2: Example Memory Map Configuration

As shown in the figure above, the first pages, containing the interrupt and exception vectors, are mapped into RAM. The next few pages are mapped into ROM for the OS `TEXT` section. The following few pages are mapped into RAM for the OS `DATA` section (copied from ROM), for the OS `bss` section (initialized

in RAM), and for OS symbols (copied from ROM). The `root` file system and resident `TEXT` segments are mapped to ROM.

If the root file system is read-only and the application needs a writable file system, the image must contain a script to create a writable RAM disk. The script should invoke the **`mkramdisk`** utility to create the writable RAM disk after the system is up and running.

See also the section "Creating Bootable Installation Media" on page 132.

KDI Build Templates

To simplify the KDI development process, LynuxWorks provides a KDI build template directory, and a tar file (*xx*`.kdi.tar.gz`) of the KDI build template, on the ODE CD-ROM. The build template and the `tar.gz` files are located in the **`kdi`** and **`tar_images`** directories, respectively.

The `kdi` directory allows users to easily build their own KDIs for execution on target hardware. A KDI build template essentially consists of all the binary and source files required to develop a custom application for the user's target system using LynxOS.

NOTE: The KDI build templates and the `tar.gz` files are not automatically installed as a part of the default installation; for information on installing KDI build templates and the `tar.gz` files, see the *LynxOS Installation Guide*.

The `/demo/demo.`*`bsp_name`* and `tar_images/`*`xx`*`.demo_`*`bsp_name`*`.tar.gz` directories have prebuilt versions of these KDIs, which can be loaded onto a user's system for evaluation and demonstration.

The sections that follow demonstrate how to set up a KDI project area, build a KDI, and download and execute it on a target. Because the project scripts are written in Bourne Shell, these build steps can be followed on any cross development or native development system.

Template Conceptual Overview

LynuxWorks provides KDI directories to give users a set of templates from which they can jump-start their own development of KDIs. The scripts automate the process of creating a project directory, copying a kernel build directory, installing

the appropriate device drivers, building the kernel, selecting the appropriate files from the LynxOS distribution, and generating a KDI.

Included KDI Build Templates

The following table describes the KDIs that are provided with LynxOS.

Build Template	Summary
`developer`	Contains development and networking utilities that provide a minimal configuration for development. Includes the following utilities: • `disk`--Disk formatting and partitioning utilities • `gnutar`--gnutar and gnuzip utilities • `shell`--Minimal shell configuration • `vm`--Virtual memory • `skdb`--Simple kernel debugger • `totaldb`--TotalDB kernel debugger • `dhcp`--DHCP protocol support (not enabled by default) • `rcp`--RCP (remote copy) utility • `ftp`--FTP (file transfer protocol) utility • `tcpdump`--Displays TCP information • `tview`--TotalView • `apache`--Apache web server • `nfsmount`--NFS mount utility The `developer` demo can be used with host system IDEs VisualLynux and CodeWarrior. The Developer KDI is documented in the section “Example--Building, Booting, and Using the developer KDI” on page 127.
`hello`	Simple configuration demonstrating a full kernel, RAM/ROM file system and a small `hello_world.cpp` application Since this is a minimal KDI, users must power down the system to end the demo.

Build Template	Summary
`install`	This KDI contains basic networking and disk utilities that can be loaded onto a target board with FTP. `install.kdi` performs these steps: 1. Determines what disk you have 2. Partitions the disk using `mkpart`. 3. Prepares a file system on the disk (if needed) and mounts it. Starts disk swapping (if needed). 4. Uses FTP to retrieve tar images. 5. Extracts tar images. 5. Makes device nodes and sets up networking.
`rlogin`	This KDI creates a simple TCP/IP configuration, demonstrating the use of `rlogin`. On the target board, boot the KDI and log in as root. Use `rlogin` to log into a system on the network. Use either the hostname (if available in `/etc/hosts`) or IP address: **`rlogin`** *<hostname | ip_address>* From another system, `rlogin` to the target.
`rootfs`	This KDI provides a basic root file system for a target board. This KDI does not contain a RAM disk file system, but boots directly from the hard disk instead. The KDI defaults to mount `/dev/ide.2a`. If the file system requires a different device, the KDI must be rebuilt to use that device.

Build Template	Summary
tkwish	This KDI demonstrates the use of the WISH shell to create an X Windows-based user interface to the target board. A simple interactive window is displayed to let the user browse several files on the file system. Use the following instructions to set up tkwish: 1. On the target board, set the DISPLAY variable to point to the X server. If there is an entry in the /etc/hosts file for the X server machine, use the hostname of the system. Otherwise, use the IP address: **DISPLAY=**<*hostname* | *ip_address*>**:0.0** **export DISPLAY** 2. On the X server, enable remote clients: **xhost +** 3. On the target board system, type demo.tkwish. 4. On the X server use the mouse to point and click.
tutorial	This KDI provides an interactive demonstration of processes and threads for LynxOS. Type **q** to quit the demo.

kdi Directory Structure

The kdi directory includes the following files and directories:

Table 8-2: KDI Directory Structure

KDI Directory/File	Description
[0-9]*	These are KDI build directories.
Makefile	This is the top-level Makefile.
PROJECT.sh	This Bourne Shell script sets up a custom KDI build environment.
README	This file describes contents of the kdi directory.
common	This directory contains shared source code, binaries and scripts.
bsp	This directory contains single port files for target hardware boards.

KDI Build Directories

Additionally, there is a distinct ordering of the KDI build subdirectories. LynuxWorks uses the naming scheme *xy.name*, according to functionality and complexity. *x*, *y*, and *name* are defined as follows:

Table 8-3: KDI Naming Convention

Convention	Description
x =0	KDIs that assist in installing LynxOS on the target
x =1	Simple KDIs that do not require networking (TCP/IP)
x =2	KDIs requiring TCP/IP
x =3	KDIs that use NFS, PPP, SNMP or other networking protocols
x =4-9	Special purpose KDIs
y =0-9, a-z, A-Z	Range of 62 possible unique identifiers
name	Any string summarizing the capability of the KDI

Restrictions

This demonstration harness assumes that a single user is building the KDI. It also assumes that the building of demo KDIs is done one at a time, since all demo KDIs share the same BSP directory where kernels are built; therefore, two KDI directories cannot be built in parallel.

Getting Started

A shell script, **`PROJECT.sh`**, is provided to automate the setup process. Before running this script, users must have already installed LynxOS from the distribution CD-ROMs onto the system's hard disk, or have mounted it from a CD-ROM; and have sourced the **`SETUP.bash`** or **`SETUP.csh`** scripts provided.

These scripts set up the **`PATH`** environment variable, and also set **`ENV_PREFIX`** to point to the distribution directory. `ENV_PREFIX` must be set correctly for `PROJECT.sh` to work.

To set up `ENV_PREFIX`, source the `SETUP.bash` or `SETUP.csh` scripts provided in the LynxOS release directory. To verify that `ENV_PREFIX` is set correctly, enter the following command, which lists the LynxOS distribution:

```
$ ls $ENV_PREFIX
```

If `ENV_PREFIX` is set correctly, then execute the `PROJECT.sh` script by entering the following command:

```
$ ENV_PREFIX/kdi/PROJECT.sh
```

The following screen output is displayed:

```
=====================================================
| Project set up script                             |
|                                                   |
| This script will create a customized directory for  |
| building and experimenting with Kernel Downloadable |
| Images (KDIs.)                                    |
=====================================================
Note: 10MB of available disk space is needed for minimal project
directory. 30MB is recommended.
Project directory location? [/tmp/newproj]
```

Enter a location on disk where the user has write permission (`/tmp/newproj` is the default) with at least 10 MB of free space available (30 MB is recommended). The following output is displayed:

```
The following BSPs are supported:
     PPC  [pc_drm,pp_drm,cpci_drm,
               vmpc,mpc860t_fads,mpc8260_vads,pmc860]
     x86  [x86_at,x86_drm,cpci_x86]
Which BSP should we use?[xxx]
Note: The default selection displayed is $ENV_PREFIX/sys.
Enter target network name and IP address now?  [y]
```

If a machine name and an IP address for the target board has been selected, enter **`y`**. The `PROJECT.sh` script goes through the process of modifying the `rc.network` file for the specified target. If **`n`** is entered, then the `rc.network` file that gets configured into the system's KDIs interactively prompts the user for the machine name and IP address at boot-time as follows:

```
What is the network name of the target board?  [lynxdemo]
Enter the target machine name:
What is IP address name of the target board?   [1.1.1.1]
Enter your target IP address.
```

Once all questions have been answered, `PROJECT.sh` proceeds to set up the customized KDI build environment, copying the appropriate files.

Building KDIs

To begin building KDI demos, change the working directory to `/tmp/newproj/11.hello`, and issue a **`make all`** command; the following steps are executed:

```
#########
# Step 1a. Modify CONFIG.TBL
```

```
#########
```

This step creates a local copy of `CONFIG.TBL`, which is used by the kernel Makefile to configure device drivers as either in or out. This `CONFIG.TBL` file is copied to the local Board Support Package (BSP) directory (where the kernel is linked) each time the user issues a **`make`** command in this `/tmp/newproj/ll.hello` directory.

```
#########
# Step 1b. Modify uparam.h
#########
```

This step creates a local copy of the `uparam.h` header file. This file is used to specify the size of user-modifiable kernel data structures and resources.

Like `CONFIG.TBL`, the `uparam.h` file is copied to the local BSP build directory each time a **`make`** command is issued in the `/tmp/newproj/ll.hello` directory.

```
#########
# Step 2: Perform any local build actions defined in desc.bsp_name.sh
#########
```

This is a hook that allows a KDI build directory to do any special local processing. Look at the **`desc.*`** file for details.

```
#########
# Step 3: Make kernel
#########
```

This step changes directory to the `./bsp.*` directory, and makes the kernel, using the `CONFIG.TBL` and `uparam.h` files from the KDI build directory.

```
#########
# Step 4: Build KDI
#########
```

This step is run from within the KDI build directory. The **`mkimage`** tool is invoked using the KDI `mkimage` specification file to pull together the appropriate kernel components and applications.

```
#########
# Step 5: Build Complete
#########
```

Look at `hello.kdi` in `/tmp/newproj`. An output similar to the following is displayed:

```
-rwxr-xr-x  1 int 774144 May 11 19:38 /tmp/newproj/hello.kdi
```

The KDI can now be downloaded and executed on the target. For more information on demo KDIs and loading KDI images onto the target, see the appropriate *LynxOS Board Support Guide*.

Example--Building, Booting, and Using the developer KDI

The Developer KDI includes development and networking components that provide a development environment on a target board. This section describes how to create, boot, and user the Developer KDI. This example uses the following configurations:

Table 8-4:

Parameter	Host	Target
Platform	Red Hat Linux 7.2 LynxOS 4.0 Cross Development `pc_680`	Force PowerCore 680
Hostname	`linuxcdk`	`fpc1`
IP Address	`192.1.1.1`	`192.1.1.2`
Ethernet Address	Not needed.	`08:00:3E:23:8C:BD`

Configuring the Developer KDI

The default configuration of the Developer KDI contains many utilities that provides a wide range of functionality for a variety of development tasks. The default configuration of the Developer KDI can change by editing the `developer.spec` file to add or remove functionality in the KDI.

Removing Unnecessary Components

The size of the Developer KDI demo may be too large for some systems. The default configuration is close to 10 MB. Unnecessary components can be removed from the default KDI image by commenting out the lines in `developer.spec` and rebuilding the kernel.

Enabling Required Components

Several components are not included in the default KDI. To enable the following components, uncomment the required lines in the `developer.spec` file:

- Linux ABI Compatibility Layer (multiple files)
- SpyKer Kernel Event Trace Analyzer (spyker target file)

- Total View
- VisualLynux
- CodeWarrior

Check the developer.spec file and uncomment any files that are required by these utilities to enable them in the Developer KDI.

Rebuilding the KDI

After editing the `developer.spec` file, rebuild the kernel.

```
# make all
```

Configuring the Linux Cross Development Host

TFTP is required by the target to load the developer.kdi. Use the following instructions to enable tftp on the host system:

1. Edit `/etc/ethers` to include the Ethernet address of the Target board:

   ```
   # vi /etc/ethers
   ```

```
08:00:3E:23:8C:BD     fpc1
```

Figure 8-3: /etc/ethers File

2. Edit the `/etc/hosts` file to include the hostname and IP address of the target board:

   ```
   # vi /etc/hosts
   ```

```
192.168.1.2      fpc1
```

Figure 8-4: /etc/hosts File

3. Create the `tftpboot` directory:

   ```
   # mkdir /tftpboot
   ```

4. Enable TFTP by editing the `tftp` file

   ```
   # cd /etc/xinetd.d/
   # vi tftp
   ```

5. In the `disable` field, type "**no**" to enable tftp.
6. In the `server_args` field type "**`/tftpboot`**". The following provides a sample `tftp` file.

```
# default: off
# description: The tftp server serves files using the trivial file transfer \
# protocol. The tftp protocol is often used to boot diskless \
# workstations, download configuration files to network-aware printers, \
# and to start the installation process for some operating systems.
service tftp
{
socket_type = dgram
protocol = udp
wait = yes
user = root
server = /usr/sbin/in.tftpd
server_args = /tftpboot
disable = no
}
```

Figure 8-5: Sample TFTP Configuration File

7. Restart the `xinetd` services:

   # **`cd /etc/rc.d/init.d`**

   # **`./xinetd restart`**

Booting the KDI

At the `PowerBoot>` prompt, type in the following command to netboot the KDI:

`PowerBoot>` **`netload developer.kdi 400000`** *<hostIP> <targetIP>*

For this example, type the following:

`PowerBoot>` **`netload developer.kdi 400000 192.168.1.1 192.168.1.2`**

NOTE: Note that if the Developer KDI will not load, it may be too large to boot. Removing components in the Developer KDI demo will allow it to boot.

Using the KDI

The Developer KDI can be used in a variety of ways. With the Apache component enabled in the developer KDI, a browser can open an `http` connection to the target. For example:

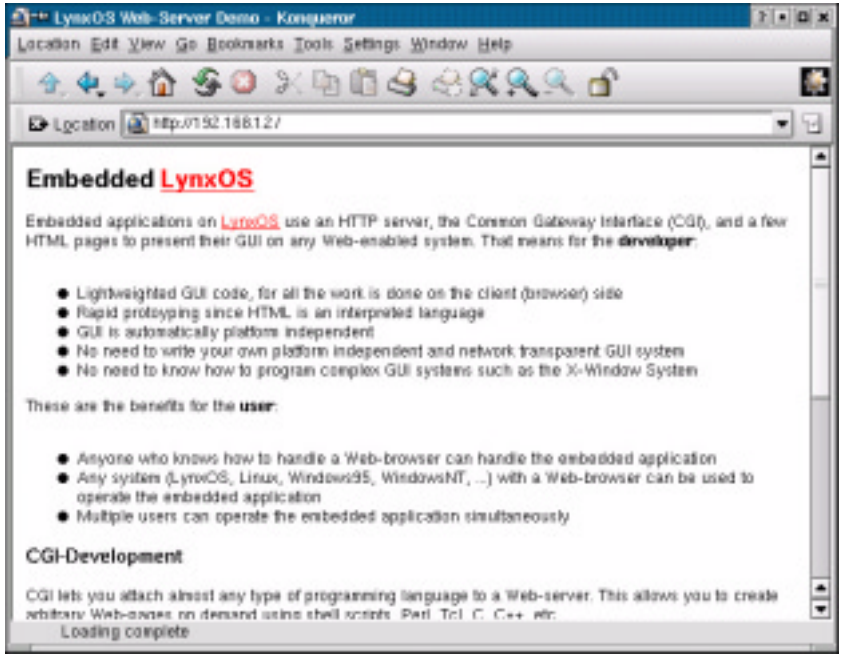

Figure 8-6: Apache Web Server Running on Target

This HTTP demonstration provides an example of the different types of development that can be done on the target.

Default Usernames and Passwords

The Developer KDI includes user accounts used to access the target board. LynuxWorks IDEs VisualLynux and CodeWarrior can connect to the target board

and perform various development tasks. The following table provides the user name and passwords for these utilities:

Table 8-5: Default Username and Passwords in the Developer KDI

Component	Username	Password
FTP	`ftp`	`ftp_user`
VisualLynux	`vl`	`vl_user`
CodeWarrior	`cw`	`cw_user`
TotalView	`tv`	`tv_user`

ROMing Issues

The tools and steps used to burn an image into a PROM are specific to the target hardware. Depending on the PROM burner and target hardware, users may be able to burn the image file as is (with no conversion to an intermediary file format) into a PROM. Refer to the specific hardware documentation for more information.

The GNU **`objcopy`** utility may be used to convert the image to the file format required by the user; see the `objcopy` man page for further information.

Generating PROM Images on x86 Systems

Building the Jump Code

BIOS limitations on the x86 architecture require that a small object module, **`rkjump`**, be used to boot up and load a LynxOS kernel from ROM. It must be available within the address range 640 KB to 1 MB at system power up. On most BIOS types, `rkjump` must reside in the address range `0xC0000` to `0xDFD00` on a `0x200` byte boundary.

After taking control from the system BIOS, the `rkjump` module initializes the Segment Descriptor Registers, sets the CPU to protected mode, and transfers control to the ROM segment containing the LynxOS image.

`rkjump` is very specific to the memory architecture and must be tuned to the specific target's needs. The sources and build environment for `rkjump` are in `/src/bin/rkjump386`. Before building the `rkjump` module, users must define

the kernel image ROM address in the `rkconfig.h` file in the `/src/bin/rkjump386` directory.

For example, if the kernel image has been created with a load address of `0x00100000`, users must specify an address of `+ 0x20` in `/src/bin/rkjump/rkconfig.h`. The `rkjump` module then needs to be rebuilt by entering the following commands:

```
$ cd /src/bin/rkjump386
$ vi rkconfig.h
```

Change the value of `RKJUMP_ADDRESS` as follows:

```
# define RKJUMP_ADDRESS  0x00100020
```

Exit the vi text editor, and enter the following commands:

```
$ touch rkjump.c
$ make all
```

Converting the Jump Code

After building the customized `rkjump` for the image being created, use the GNU `objcopy` utility to prepare the file for transfer to ROM.

Converting the Kernel Image

Convert the kernel image created with `mkimage` into hex format using the GNU `objcopy` utility.

Creating Bootable Installation Media

LynxOS facilitates creating bootable installation floppy disks for x86 platforms, and bootable CD-ROMs for x86 and PowerPC platforms; “Creating a Bootable x86 Floppy” and “Creating a Bootable x86 or PowerPC CD-ROM” below detail this, respectively.

Creating a Bootable x86 Floppy

LynxOS users can create bootable LynxOS installation floppies for x86 systems on formatted 1.44 MB disks (for more information on formatting floppies, see "Formatting Floppy Disks" on page 48 and the `fmtflop` man page).

To create a bootable LynxOS installation floppy disk, perform the following steps:

1. Place a formatted 1.44 MB floppy disk in the x86 system's floppy drive.
2. Run the build script by entering the following command line:

   ```
   $ /sys/romkit/installation/Build
   ```

3. Follow the on-screen instructions.

   The `/sys/romkit/installation/Build` script assumes that no file systems are mounted on the `/mnt` directory. Users need be sure that `/mnt` is available for use before using the script. If a file system is mounted on `/mnt`, the following command unmounts it:

   ```
   $ umount /mnt
   ```

For more information on mounted file systems, see the **`mount`** man page.

Creating a Bootable x86 or PowerPC CD-ROM

LynxOS users can create a bootable LynxOS installation CD-ROM for either x86 or PowerPC systems.

To create a LynxOS installation CD-ROM for either x86 or PowerPC platforms, perform the following steps *as root*:

Creating CD-ROM Image on Host

1. Create a directory on a file system partition that has enough free disk space to unload all of the desired `tar.gz` image files by entering the following command:

   ```
   $ mkdir -r /media/tar_images
   ```

   where *`media`* is the desired name for the bootable CD-ROM.

2. Change directory to the newly created directory:

   ```
   $ cd /media/tar_images
   ```

3. Locate the desired `tar.gz` image files (originally in the `/tar_images` directory on the ODE CD-ROM). The following are the required `tar.gz` image files:

   xxxxx`.devos.tar.gz`
   xxxxx`.bsp_`*bsp_name*`.tar.gz`

   bsp_name is the type of Board Support Package (BSP) and *xxxxx* is the LynuxWorks naming scheme for the KDI build subdirectories. (See "KDI Build Directories" on page 124.)

4. Copy the desired `tar.gz` image files into the */media*`/tar_images` directory:

   ```
   $ cp xxxxx.devos.tar.gz
   xxxxx.bsp_bsp_name.tar.gz /media/tar_images
   ```

5. Extract the `tar.gz` image files into the */media* directory by entering the following `tar` commands:

   ```
   $ tar zxpf tar_images/xxxxx.devos.tar.gz
   $ tar zxpf tar_images/xxxxx.bsp_bsp_name.tar.gz
   ```

   In the */media*`/sys` directory, create a symbolic link from `bsp.`*bsp_name* to `lynx.os`:

   ```
   $ cd /media/sys
   $ ln -s bsp.bsp_name lynx.os
   ```

6. Copy the `nodetab` file from */media*`/sys/lynx.os` to */media*`/etc`:

   ```
   $ cp lynx.os/nodetab ../etc
   ```

7. Change the working directory to */media*`/dev` and create device nodes in */media*`/dev` with the following command:

   ```
   $ cd /media/dev
   $ mknod -a ../etc/nodetab
   ```

8. Change the working directory back to */media* and copy the kernel file located in */media*`/sys/lynx.os/a.out` to */media*`/lynxos`:

   ```
   $ cd /media
   $ cp sys/lynx.os/a.out lynx.os
   ```

Creating the CD-ROM

1. Ensure that the CD-ROM is read as bootable by entering the following command (where *platform* is one of either x86 or ppc):

   ```
   $ touch tar_images/
   .this_is_a_bootable_platform_CD
   ```

2. Create the ISO formatted image using the **mkisofs** command (see the mkisofs man page for further information):

   ```
   $ mkisofs -R -b preboot ../iso_image .
   ```

3. Change the working directory up one level:

   ```
   $ cd  ..
   ```

4. Use the **makeboot** utility to configure the iso_image file:

   ```
   $ makeboot -r major,minor -b device
   -flag autoboot_CD -timeout secs iso_image
   ```

NOTE: makeboot must be run on the platform from which iso_image is to be booted (see the makeboot man page for more information).

---

5. Use the **cdwrite** utility to write the bootable iso_image to a recordable CD-ROM (see the cdwrite man page for further information) by entering the following command:

   ```
   $ cdwrite -i iso_image -r -v -s speed cdr_device
   ```

NOTE: The command **-s** *speed* sets the speed factor used for writing the CD-ROM.The default speed is 1. This means a data transfer rate of 150 KB per second. It takes 74 minutes to fill a 650 MB CD-ROM at speed factor 1.

The approximate times to write lead-in and lead-out tracks are as follows:

- -s 1 4 minutes
- -s 2 2 minutes
- -s 4 1 minute

CHAPTER 9 *Linux ABI Compatibility*

The Linux ABI (Application Binary Interface) compatibility feature of LynxOS allows Linux binary applications to run under LynxOS. This chapter provides a detailed overview of the Linux ABI feature.

Overview

LynxOS supports executing dynamically-linked Linux binary applications on LynxOS systems as if they were native LynxOS applications. There is no need to rebuild Linux applications with LynxOS tools, or even access the source code. Linux application binaries can be installed and executed on a LynxOS machine in the same manner as they are installed and executed on a Linux system. The Linux ABI feature adds a new level of flexibility by allowing users to use both Linux and LynxOS binaries in parallel on a single LynxOS system.

Linux ABI compatibility is made possible by adding a Linux ABI Layer that includes Linux libraries. “Native” LynxOS applications (applications built for LynxOS) are unaffected by the addition of the Linux ABI Layer.

Installing the Linux ABI Layer

The Linux ABI Layer is installed from the Additional Components CD-ROM. Refer to the *LynxOS Installation Guide* for installation instructions.

In addition to the basic set of standard Linux shared libraries (`linuxabi.tar.gz`), LynuxWorks provides a more comprehensive set of standard Linux shared libraries (`linuxabi_advanced.tar.gz`) for users who require libraries that may not be included in the basic set. Note that this file is See the *LynxOS Installation Guide* for more details on installing these libraries.

Linux ABI Layer

This section provides a brief overview of the Linux ABI software layer. The following diagram shows how a Linux binary runs on LynxOS under the Linux ABI Layer:

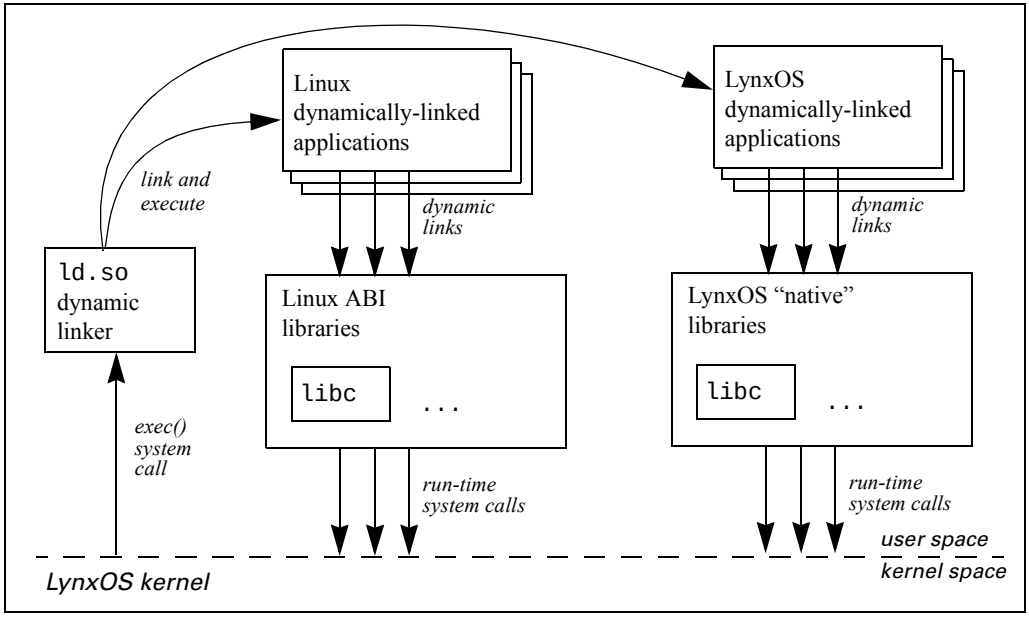

Figure 9-1: Linux ABI Software Layer

A dynamically-linked application (built for either LynxOS or Linux), relies on the `ld.so` dynamic linker and loader to complete the process of linking all necessary references to shareable objects before the application is executed.

Linux application binaries are linked by `ld.so` into the shared libraries that compose the Linux ABI Layer. This Linux ABI layer translates Linux application shared object interface calls into calls that are binary-compatible with LynxOS kernel interfaces. Typically, implementation of standard interfaces are similar enough between Linux and LynxOS that only a simple translation (or none at all) is required before a call from a Linux application can go into the LynxOS kernel.

Interoperability with LynxOS Native Applications

The Linux ABI Layer relies on the LynxOS `ld.so` dynamic linker to resolve Linux application calls into shared objects using an appropriate set of Linux shared libraries. Native LynxOS applications are not affected by the Linux ABI feature in any way. LynxOS native applications continue to be linked into the LynxOS native shared libraries, and function as before.

Linux ABI Shared Libraries

The following table shows the shared libraries included in the Linux ABI Layer. The libraries are located in the same directories as they are on Linux (specified in the `DT_RPATH` field of the application ELF header). This allows the LynxOS `ld.so` dynamic linker and loader to link the Linux binary into the Linux ABI shared libraries.

Table 9-1: Linux ABI Shared Libraries

x86 Shared Libraries	PowerPC Shared Libraries
lib/ lib/libc.so.6 lib/libcrypt.so.1 -> libcrypt-2.2.2.so lib/liblynxpthread.so lib/libm.so.6 -> libm-2.2.2.so lib/libnsl.so.1 -> libnsl-2.2.2.so lib/libpthread.so.0 lib/librt.so.1 lib/libtermcap.so.2 -> libtermcap.so.2.0.8 lib/libutil.so.1 -> libutil-2.2.2.so lib/ld-linux.so.2 -> ../usr/lib/ld.so.1 lib/libresolv.so.2 -> libresolv-2.2.2.so lib/libcrypt-2.2.2.so lib/libnsl-2.2.2.so lib/libtermcap.so.2.0.8 lib/libutil-2.2.2.so lib/libresolv-2.2.2.so lib/libdl.so.2 -> shlib/libdl.so lib/libm-2.2.2.so	lib/ lib/libc.so.6 lib/libcrypt.so.1 -> libcrypt-2.2.1.so lib/liblynxpthread.so lib/libm.so.6 -> libm-2.2.1.so lib/libnsl.so.1 -> libnsl-2.2.1.so lib/libpthread.so.0 lib/librt.so.1 lib/libtermcap.so.2 -> libtermcap.so.2.0.8 lib/libutil.so.1 -> libutil-2.2.1.so lib/ld.so.1 -> ../usr/lib/ld.so.1 lib/libdl.so.2 -> shlib/libdl.so lib/libncurses.so.4 lib/libcrypt-2.2.1.so lib/libm-2.2.1.so lib/libnsl-2.2.1.so lib/libutil-2.2.1.so lib/libtermcap.so.2.0.8 lib/libresolv-2.2.1.so lib/libresolv.so.2 -> libresolv-2.2.1.sousr/
usr/X11R6/ usr/X11R6/lib/ usr/X11R6/lib/libICE.so.6 -> libICE.so.6.3 usr/X11R6/lib/libSM.so.6 -> libSM.so.6.0 usr/X11R6/lib/libX11.so.6 -> libX11.so.6.2 usr/X11R6/lib/libXext.so.6 -> libXext.so.6.4 usr/X11R6/lib/libXmu.so.6 -> libXmu.so.6.2 usr/X11R6/lib/libXt.so.6 -> libXt.so.6.0 usr/X11R6/lib/libXaw.so.7 -> libXaw.so.7.0 usr/X11R6/lib/libXft.so.1 -> libXft.so.1.0 usr/X11R6/lib/libXpm.so.4 -> libXpm.so.4.11 usr/X11R6/lib/libXrender.so.1 -> libXrender.so.1.0 usr/X11R6/lib/libICE.so.6.3 usr/X11R6/lib/libSM.so.6.0 usr/X11R6/lib/libX11.so.6.2 usr/X11R6/lib/libXext.so.6.4 usr/X11R6/lib/libXmu.so.6.2 usr/X11R6/lib/libXt.so.6.0 usr/X11R6/lib/libXaw.so.7.0 usr/X11R6/lib/libXft.so.1.0 usr/X11R6/lib/libXpm.so.4.11 usr/X11R6/lib/libXrender.so.1.0 usr/X11R6/lib/libXi.so.6 -> libXi.so.6.0 usr/X11R6/lib/libXi.so.6.0	usr/X11R6/ usr/X11R6/lib/ usr/X11R6/lib/libICE.so.6 -> libICE.so.6.3 usr/X11R6/lib/libSM.so.6 -> libSM.so.6.0 usr/X11R6/lib/libX11.so.6 -> libX11.so.6.2 usr/X11R6/lib/libXext.so.6 -> libXext.so.6.4 usr/X11R6/lib/libXmu.so.6 -> libXmu.so.6.2 usr/X11R6/lib/libXt.so.6 -> libXt.so.6.0 usr/X11R6/lib/libXaw.so.6 -> libXaw.so.6.1 usr/X11R6/lib/libX11.so.6.2 usr/X11R6/lib/libXaw.so.6.1 usr/X11R6/lib/libXaw.so.7 -> libXaw.so.7.0 usr/X11R6/lib/libXaw.so.7.0 usr/X11R6/lib/libXext.so.6.4 usr/X11R6/lib/libXmu.so.6.2usr/X11R6/lib/ libXt.so.6.0 usr/X11R6/lib/libICE.so.6.3 usr/X11R6/lib/libSM.so.6.0 usr/X11R6/lib/libXft.so.1.0 usr/X11R6/lib/libXft.so.1 -> libXft.so.1.0 usr/X11R6/lib/libXrender.so.1.0 usr/X11R6/lib/libXrender.so.1 -> libXrender.so.1.0

Table 9-1: Linux ABI Shared Libraries

x86 Shared Libraries	PowerPC Shared Libraries
usr/lib/libstdc++-libc6.1-1.so.2 -> libstdc++-2-libc6.1-1-2.9.0.so usr/lib/libutempter.so.0 -> libutempter.so.0.5.2 usr/lib/libbz2.so.1 -> libbz2.so.1.0.0 usr/lib/libfreetype.so.6 -> libfreetype.so.6.0.1 usr/lib/libreadline.so.4.1 usr/lib/libstdc++-libc6.2-2.so.3 -> libstdc++-3-libc6.2-2-2.10.0.so usr/lib/libstdc++-2-libc6.1-1-2.9.0.so usr/lib/libutempter.so.0.5.2 usr/lib/libbz2.so.1.0.0 usr/lib/libfreetype.so.6.0.1 usr/lib/qt-2.3.0/ usr/lib/qt-2.3.0/lib/ usr/lib/qt-2.3.0/lib/libqt.so.2 -> libqt.so.2.3.0 usr/lib/qt-2.3.0/lib/libqt.so.2.3.0 usr/lib/libjpeg.so.62 -> libjpeg.so.62.0.0 usr/lib/libpng.so.2 -> libpng.so.2.1.0.9 usr/lib/libz.so.1 -> libz.so.1.1.3 usr/lib/libGLU.so.1 -> libGLU.so.1.1.030401 usr/lib/libGL.so.1 -> libGL.so.1.2.030401 usr/lib/libncurses.so.5 -> libncurses.so.5.2 usr/lib/libncurses.so.5.2 usr/lib/libmng.so.1.0.0 usr/lib/libmng.so.1 -> libmng.so.1.0.0 usr/lib/libGL.so.1.2.030401 usr/lib/libGLU.so.1.1.030401 usr/lib/libz.so.1.1.3 usr/lib/libz.so -> libz.so.1.1.3 usr/lib/libpng.so.2.1.0.9 usr/lib/libjpeg.so.62.0.0 usr/lib/libreadline.so.4 -> libreadline.so.4.1 usr/lib/libstdc++-3-libc6.2-2-2.10.0.so	usr/lib/libstdc++-libc6.1-1.so.2 usr/lib/libutempter.so.0 -> libutempter.so.0.5.2 usr/lib/libstdc++-libc6.1-2.so.3 -> libstdc++-3-libc6.1-2-2.10.0.so usr/lib/libncurses.so.5 -> libncurses.so.5.2 usr/lib/libncurses.so.5.2 usr/lib/libreadline.so.4 -> libreadline.so.4.1 usr/lib/libreadline.so.4.1 usr/lib/libstdc++-2-libc6.1-1.1-2.9.0.so usr/lib/libstdc++-libc6.1-1.1.so.2 -> libstdc++-2-libc6.1-1.1-2.9.0.so usr/lib/libutempter.so.0.5.2 usr/lib/libreadline.so.3.0 usr/lib/libstdc++-3-libc6.1-2-2.10.0.so usr/lib/libbz2.so.1 -> libbz2.so.1.0.0 usr/lib/libbz2.so.1.0.0 usr/lib/libjpeg.so.62 usr/lib/libpng.so.2.1.0.9 usr/lib/libpng.so.2 -> libpng.so.2.1.0.9 usr/lib/qt-2.3.1/ usr/lib/qt-2.3.1/lib/ usr/lib/qt-2.3.1/lib/libqt.so.2.3.1 usr/lib/qt-2.3.1/lib/libqt.so.2 -> libqt.so.2.3.1 usr/lib/libz.so.1 -> libz.so.1.1.3 usr/lib/libz.so.1.1.3 usr/lib/libmng.so.0.0.9 usr/lib/libmng.so.0 -> libmng.so.0.0.9 usr/lib/libstdc++-3-libc6.2-2-2.10.0.so usr/lib/libstdc++-libc6.2-2.so.3 -> libstdc++-3-libc6.2-2-2.10.0.so usr/lib/libfreetype.so.6.0.1 usr/lib/libfreetype.so.6 -> libfreetype.so.6.0.1

Adding Linux Shared Libraries to LynxOS

Additional shared libraries not included in the LynxOS Linux ABI distribution may be required in order to run certain Linux applications. For instance, the PERL library extensions are needed to run the Linux-built `perl` binary on LynxOS. Such additional libraries must be copied by the user from a Linux system and installed in the appropriate directory on LynxOS. The Linux shared libraries do not need to be rebuilt.

Determining Linux Application Library Dependencies

The shared libraries required by a particular Linux application can be viewed with the **ldd** command on a Linux system. For example, the Linux **ls** command requires the following libraries:

```
# ldd -d /bin/ls
 libtermcap.so.2 => /lib/libtermcap.so.2 (0x4002d000)
 libc.so.6 => /lib/i686/libc.so.6 (0x40031000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
# ls -l /lib/libtermcap.so.2
lrwxrwxrwx 1 root root 19 Jan 26 2002
/lib/libtermcap.so.2 ->  libtermcap.so.2.0.8
```

Updating Linux ABI Layer Libraries

Linux shared libraries can easily be updated to accommodate the needs of particular Linux applications. New libraries can easily be added to the system simply by copying them from Linux to the appropriate directories on a LynxOS system. Updated versions of shared libraries can also be added to the system by copying over the current shared library.

However, it is important to note that users should copy over the actual reference file, and not symbolic links to the reference file. For example, users should not copy over the file `libncurses.so.5.2` and rename it `libncurses.so.5`. The Linux convention is to create a symbolic link `linbncurses.so.5` to the updated library `libncurses.so.5.2`. This allows any application that calls `libncurses.so.5` to access the updated shared libraries.

```
# cp <path>libncurses.so.5.2 /usr/lib
# cd /usr/lib
# ln -s libncurses.so.5.2 libncurses.so.5
# ls -l /usr/lib/libncurses.so.5*
total 509
lrwxrwxrwx 1 root 17 Feb 22 20:00 libncurses.so.5@ ->
libncurses.so.5.2
-rwxr-xr-x 1 root 257524 Feb 19 18:05 libncurses.so.5.2*
```

Linux ABI Shared Libraries that Should Not Be Overwritten

User should not overwrite the following shared libraries installed from the Linux ABI distribution:

- `/lib/libpthread.so.*`
- `/lib/librt.so.*`
- `/lib/libc.so.*`
- `/lib/ld.*`

These are LynxOS-specific shared libraries that have the same names as Linux shared libraries. These LynxOS files are vital for the Linux ABI feature to function. If these LynxOS files are accidentally replaced by the Linux-based shared libraries, the Linux ABI environment will not function.

Specifying Linux ABI Shared Library Paths

The `LD_LIBRARY_PATH` environment variable specifies a search path for ELF shared libraries used by the `ld.so` dynamic linker. To ensure that `ld.so` locates all Linux ABI libraries required for a particular Linux application, `LD_LIBRARY_PATH` must include the path names to all the locations (directories) where respective Linux ABI libraries are located. `LD_LIBRARY_PATH` must be set up and exported before a Linux application can be executed.

NOTE: If the LynxOS X & Motif package is installed Before running a Linux GUI application, it is important to specify the `LD_LIBRARY_PATH` search paths in the correct order. Note that the LynxOS X11 shared libraries symbolic links are located in `/usr/lib/`, whereas Linux X11 libraries are located in `/usr/XllR6/lib`, Before running any Linux GUI application, it is necessary to define `LD_LIBRARY_PATH` in the order that path `/usr/X11R6/lib` precedes the path `/usr/lib`, so that the Linux GUI application started up with the Linux X shared libraries. Otherwise, the Linux GUI application will crash if it starts with the LynxOS X shared libraries .

When setting the `LD_LIBRARY_PATH`, be sure to specify the Linux library paths in correct order.

The following example shows how to set up `LD_LIBRARY_PATH` for the Linux `telnet` binary (assuming that the Linux `telnet` binary is installed in `/linux/bin`):

- the Linux telnet utility:

```
$ export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
$ /linux/bin/telnet
```

- the x86 Linux Opera utility:

```
$ export LD_LIBRARY_PATH=/lib:/usr/X11R6/lib: \
/usr/lib/qt-2.3.0:/usr/lib
$ /usr/bin/opera
```

NOTE: As defined by the ELF specification, if an application has the `S_ISUID` or `S_ISGID` bits set in its protection mode, the `ld.so` dynamic linker ignores the `LD_LIBRARY_PATH` environment variable. To run such applications, the S-bits must be removed from the protection mode and the application must be run from a `root` (superuser) login.

Running Linux Applications

This section discusses additional features and functions of the Linux ABI Compatibility Layer.

Linux Reference Distribution

The term “Linux Reference Distribution” is used to identify a particular version of Linux that is compatible with the Linux ABI Layer on LynxOS. The Linux Reference Distribution is composed of specific versions of these components:

- Linux kernel
- Linux `glibc` library

The Linux Reference Distribution is used to validate and test Linux ABI Compatibility. Applications built on a supported version of the Linux Reference

Distribution are supported. As of this printing, Linux applications built on the following Linux Reference Distribution are supported:

- Linux Kernel 2.4.x
- Linux `glibc` library 2.2.2

Refer to the *LynxOS Release Notes* for updates or changes to the supported Linux Reference Distribution.

Support for Dynamically Linked Applications

The Linux ABI implementation relies on the dynamic linker (`ld.so`) to resolve calls made by Linux binaries into the shared libraries that comprise the Linux ABI library. Such resolution is performed at run-time, when an application is loaded and linked by `ld.so`.

NOTE: Execution of statically linked Linux binaries is not supported by LynxOS. This, however, is a minor limitation, because most Linux binaries are dynamically linked. Unless explicitly specified via a special option to **`ld`** during compilation, all Linux binaries require further linking at run time by `ld.so`.

Support for Versioned Symbols

Linux binaries built on earlier releases of a supported Linux distribution should function when run on LynxOS. For example, if Red Hat 7.1 is the supported Linux reference distribution, applications built on Red Hat 6.x will run on LynxOS. This is because the LynxOS `ld.so` dynamic linker supports resolution of versioned symbols in a shared library.

The Linux ABI libraries (such as the `libc` library) contain multiple, versioned definitions of an interface entry point. These multiple definitions provide backward binary compatibility for earlier releases of Linux. A defined version of an interface corresponds to a particular version of the library, and thus to a particular release of Linux. When `ld.so` resolves links into a library, the version information is available with each unresolved symbol. This allows `ld.so` to determine the version of a symbol that the application requires. By linking into an appropriate version of symbols, `ld.so` ensures that there are no version-dependent discrepancies between the definition of the interfaces and the shared libraries.

Exceptions and Limitations

Certain Linux binaries are not supported by the Linux ABI Layer. The following table provides a list of features that are not supported with the Linux ABI Layer.

Table 9-2: Exceptions and Limitations

Exception	Comments
Statically linked applications	Linux ABI Layer supports execution of dynamically linked Linux binaries only.
Applications built in a Linux context later than the Linux reference context	Linux ABI libraries contain only version of symbols for the current reference context, and previous versions of the reference context.
Applications that make direct calls into the kernel	Linux ABI Layer relies on the ABI libraries to translate calls between Linux applications and the LynxOS kernel.
Applications that uses a feature of the Linux kernel not available in the LynxOS kernel	An example of this type of an exception is an application that uses the `/proc` file system.

Extracting RPMs with rpm2cpio

Linux binaries are sometimes distributed as RPM files. Because LynxOS does not support RPM, users must manually extract the contents of the RPM file on the Linux system. The Linux utility `rpm2cpio` extracts the contents of the RPM file, which can then be piped to `cpio`:

```
# rpm2cpio <rpm_filename> | cpio -ivd
```

NOTE: Note that because `rpm2cpio` is a statically-linked binary, it cannot run on the LynxOS Linux ABI Layer.

For more information on using `rpm2cpio` and `cpio`, see the respective man pages.

Example -- Running Opera

The following example provides instructions for running the Opera Web Browser on a LynxOS x86 system with the Linux ABI Layer.

Installing Linux ABI Layer

1. On the LynxOS system, mount the Additional Components CD-ROM to an available mount point. For example,

   ```
   # mount /dev/<cdrom> /mnt
   ```

   Where *<cdrom>* is the device node of the CD-ROM drive, for example: `ide.1`.

2. Install the Linux ABI Layer:

   ```
   # cd /
   # tar xvfz /mnt/tar_images/<num>linuxabi.tar.gz
   ```

Downloading Opera

1. On the Linux system, download Opera from `http://www.opera.com`. Download the following version of the Opera Web browser:
   - Operating system: Linux (x86)
   - Language: English (United States)
   - Version: 5.0
   - Option: Qt dynamically linked
   - Package: RPM for RedHat 7.1

NOTE: Users must download the Opera web browser built for Red Hat 7.1. Other versions of Opera built for other Linux distributions may not work correctly.

Also, be sure to download the dynamically-linked version. The statically-linked binary version of Opera will not work with the Linux ABI Layer.

---

2. Download the Opera RPM file to a user's home directory: *<user_dir>*.

3. Make a directory for Opera and copy the RPM to that directory.

```
# cd <user_dir>
# mkdir opera_demo
# cd opera_demo
```

4. Use the **rpm2cpio** command to extract the contents of the RPM file:

```
# rpm2cpio ../opera-dynamic-rh71-5.0-3.i386.rpm | cpio -ivd
```

   Refer to the `rpm2cpio` and `cpio` man pages for additional information on these commands.

   The Opera binary and support files are extracted to the current directory.

5. Create a tar archive of the extracted Opera files to transfer to the LynxOS system:

```
# tar cvf opera_demo.tar ./usr*
# ls -l opera_demo.tar
-rw-r--r-- 1 root root 4454400 Feb 28 10:41 opera_demo.tar
```

6. Transfer the `opera_demo.tar` file to the LynxOS system via FTP or RCP.

7. On the LynxOS system, extract the `opera_demo.tar` file:

```
# cd <user_dir>
# mkdir opera_demo
# cd opera_demo
# tar xvf ../opera_demo.tar
```

   The Opera Linux Binary and support files are extracted to the current directory.

Configuring the Linux ABI Layer

1. Set the `LD_LIBRARY_PATH` to include the library paths required by Opera:

   - For x86,

```
# export LD_LIBRARY_PATH=/lib:/usr/X11R6/lib: \
/usr/lib/qt- 2.3.0/lib:/usr/lib
```

- For PowerPC,

```
# export LD_LIBRARY_PATH=/lib:/usr/X11R6/lib: \
/usr/lib/qt- 2.3.1/lib:/usr/lib
```

NOTE: It is important to specify the `LD_LIBRARY_PATHS` search paths in the correct order. Because LynxOS X11 libraries are located in `/usr/lib/X11`, and Linux X11 libraries are located in `/usr/XllR6`, specifying the incorrect path first can result in the Linux binary crashing. Setting the `/usr/lib` directory before `/usr/X11R6/lib` causes the LynxOS X11 libraries to be used instead of Linux libraries.

When setting the `LD_LIBRARY_PATH`, be sure to specify the Linux library paths correctly.

---

2. Set the `DISPLAY` variable to display to a local X server (if X and Motif are installed), or display to a remote X server:

   A)For a Local X server:

   ```
   # export DISPLAY=0.0
   ```

   B)For a Remote X server:

   ```
   # export DISPLAY=<Xserver_IP_address>:0.0
   ```

   Where *<Xserver_IP_address>* is the IP address of the X server. On the remote X server, enable remote X access with the **xhost** command:

   ```
   # xhost +<LynxOS_IP_address>
   ```

   Where *<LynxOS_IP_address>* is the IP address of the LynxOS system.

3. On the LynxOS system, start opera:

   ```
   # <user_dir>/opera_demo/usr/bin/opera
   ```

Figure 9-2: Opera Web Browser Running on LynxOS

CHAPTER 10 *Event Logging*

The Event Logging System

Event Logging provides the ability to log specific kernel and user application events. With the LynuxWorks Event Logging System, users can maintain system performance, predict and eliminate problems, and reduce system downtime.

Events can be monitored from several system facilities (mail, kernel, or file system, for example) with a variety of severities (warning, emergency, or critical, for example). Process threads can be set up to log specific events, monitor facility conditions and receive notifications on specific events.

Event Logging Components

The LynuxWorks Event Logging System is comprised of several components that interact to provide a fully-featured means of logging system events. The following table describes the components of the Event Logging System.

Table 10-1: Event Logging System Components

Component	Description
User Libraries	User libraries include the function calls that interact with the Event Logging System. For function prototypes and descriptions, see “The Event Logging Daemon” on page 157.
System Log Files	System Log Files include the Active System Log and Archived System Log. At any given time, the Event Logging System writes to an open file, known as the Active System Log. This log file is written to until it reaches maximum capacity. When the Active System Log file is full, it is archived and a new Active System Log is created.

Table 10-1: Event Logging System Components (Continued)

Component	Description
Event Log Driver	The Event Log Driver contains the read and write interfaces used by the Log Daemon to read and write to memory. The Event Log Driver can be installed statically or dynamically.
System Log Buffer	The System Log Buffer is a buffer of rows. Each row contains two elements: an Event Record and the Event Data. The Event Record is used to identify the contents of the User Data. The Event Data is the raw data of the event. For more information on Event Record Identifier types, see “Event Record Identifier (log_event_t)” on page 161.
Log Daemon	The Log Daemon is a user space application that maintains event notifications, and the Active System Log.

Process Flow

The following sections describe the various flows used by the reading, writing, and opening functions.

Write Flow

A user thread initiates `log_write()`, which calls the write interface of the Event Log Driver. The Event Log Driver fills the Event Record and Event Data with the parameters provided in `log_write()`. If the Log System Buffer is full, an error is returned. If successful, the Log Daemon reads the contents of the System Log Buffer and writes the event to the Active System Log. The Event Log Driver clears out the buffer for future use.

After the event is processed in the Active System Log, the Log Daemon searches the notification list to determine if any processes must access the event. If so, the process is sent a notification signal.

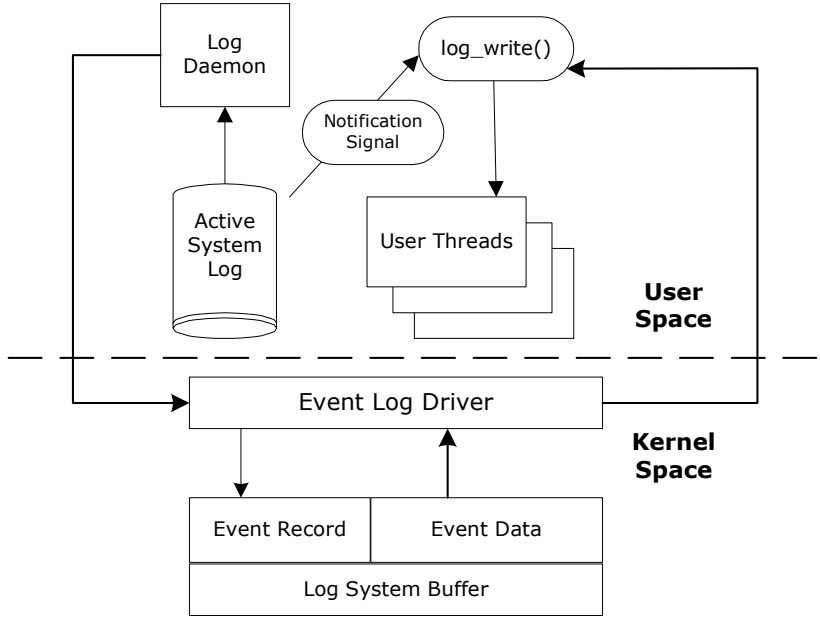

Figure 10-1: Write Flow

Read Flow

A user thread initiates a `log_read()` function, which reads the log entry of the file handle provided by `log_read()`. The function checks if the event record matches the current query requirements. If it matches, the event record data and event data are returned to the calling thread. If not, the event is skipped and the next event is checked.

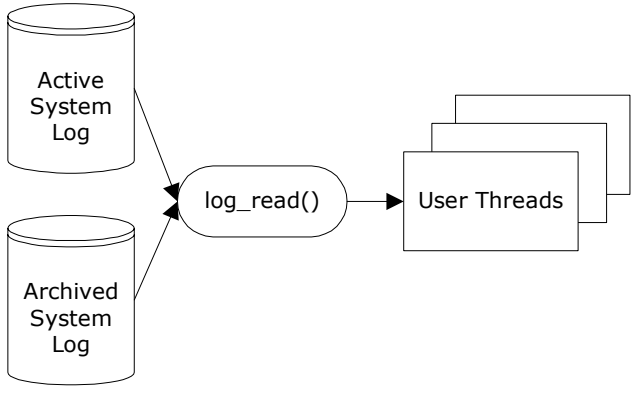

Figure 10-2: Read Flow

Open Flow

A user thread initiates the `log_open()` function, which opens a log file.

If the path parameter of `log_open()` is `NULL`, it retrieves the Active System Log file name from memory. Otherwise, the `path` parameter is used to open an Archived System Log.

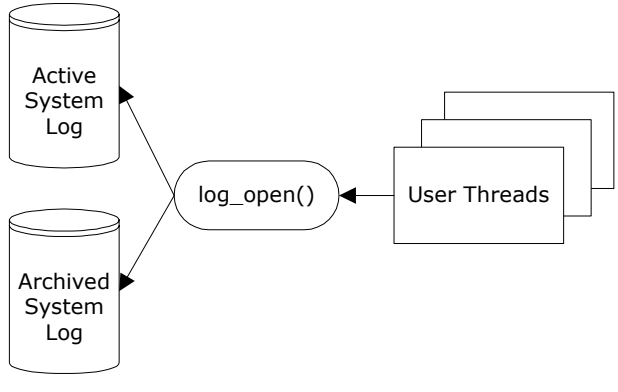

Figure 10-3: Open Flow

Notify Flow

A user thread initiates the `log_notify()` function, which sends process and signal information to the Log Daemon. If the Log Daemon finds a matching event, a signal is sent to the calling thread.

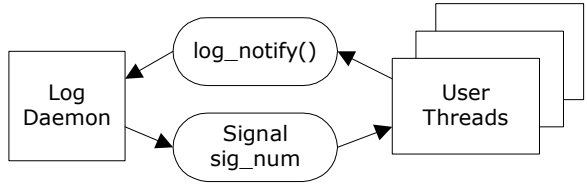

Figure 10-4: Notification Flow

Installing the Event Logging System

Installing on a Native LynxOS System

Use the following commands to install the Event Logging driver on a native LynxOS system:

```
# cd /sys/lynx.os
# make install.evlog
```

Removing from a Native System

Use the following commands to remove the Event Logging driver from a Native LynxOS system:

```
# cd /sys/lynx.os
# make uninstall.evlog
```

Installing On a Cross Development System

Use the following commands to install the Event Logging driver on a cross development system:

```
# cd /usr/lynx/4.0.0/<cpu>
```

where `<cpu>` is the platform of the system, `x86` or `ppc`.

```
# source SETUP.bash
# cd $ENV_PREFIX/sys/lynx.os
# make install.evlog
```

Removing From a Cross Development System

Use the following commands to remove the Event Logging driver from a cross development system.

```
# cd /usr/lynx/4.0.0/<cpu>
```

where `<cpu>` is the platform of the system, `x86` or `ppc`.

```
# source SETUP.bash
# cd $ENV_PREFIX/sys/lynx.os
# make uninstall.evlog
```

Configuring the Event Logging System

Event Logging Parameters

The following sections provide the configurable parameters available to the Event Logging System. Depending on the installation type, these options are set in the `evloginfo.h` header file.

Configurable Parameters

Buffer Length

The default buffer length for the Event Logging System is 80. The maximum, defined as `LOG_ENTRY_MAXLEN` is 250.

Buffer Size

The buffer size represents the number of rows the Log System Buffer contains. The default value is 100.

High Watermark Level

The High Watermark Level is the percentage of how full a Log System Buffer is before a high watermark message is written. The default High Watermark Level is 90%. After the Log System Buffer reaches the High Watermark Level, a High Watermark Level record is written to the buffer stating that this level has been reached. This system message has a facility `LOG_EVLOG` and severity `LOG_WARNING`.

The High Watermark Level can be disabled by setting it to 100.

When one entry remains in the Log System Buffer, a message is written to the last buffer stating that the Log System Buffer is full. Additional events cannot be written to the buffer. `log_write()` fails when the System Buffer is full.

Low Watermark Level

The Low Watermark Level is the percentage of how empty a Log System Buffer is before a low watermark message is written. The default Low Watermark Level is 0%. When the Log System Buffer meets the Low Watermark Level, a log entry is written to the buffer stating the Low Watermark Level is reached. This system message has a facility of `LOG_EVLOG` and a severity of `LOG_WARNING`.

The Event Logging Daemon

The Event Logging Daemon, **`evlogd`** must be running to use the Event Logging System. To start the Event Logging Daemon, type `evlogd`:

```
# evlogd
```

`evlogd` can also be added to the `/bin/rc` file to start the Event Logging Daemon when the system starts.

NOTE: The Event Logging driver must be installed before the Event Logging Daemon can run. See “Installing on a Native LynxOS System” on page 155.

`evlogd` has the following command line options:

Table 10-2: Event Log Daemon Options

Command Option	Description
`-l <number_of_entries>`	The **`-l`** option specifies the number of log entries written to the Active System Log before creating a new log file. The default value is 10000. This value can be changed from a minimum of 10 and a maximum of 999999.
`-d <path>`	The **`-d`** option specifies a path to the log files on a system. The default path is `/sys/log`. The directory name cannot exceed 80 characters.

Function Prototypes and Descriptions

typedefs in eventlog.h

Log File Descriptor (logd_t)

The `logd_t` type represents a log file stream opened for reading. The `logd_t` type is defined as an unsigned short.

System Identifier (log_facility_t)

The `log_facility_t` type identifies the subsystem that generated the event. The following table lists the types, memory addresses and descriptions of the available facilities.

Table 10-3: log_facility_t Values

Name	Bit Address	Description
`LOG_AUTH`	0x00000001	Authorization & authentication events
`LOG_CRON`	0x00000002	`cron` daemon

Table 10-3: log_facility_t Values (Continued)

Name	Bit Address	Description
LOG_DAEMON	0x00000004	Background services
LOG_LPR	0x00000008	lpr subsystem
LOG_MAIL	0x00000010	Mail delivery system
LOG_NEWS	0x00000020	News system
LOG_KERN	0x00000040	General Kernel events
LOG_KERN_FS	0x00000080	File system events
LOG_KERN_NET	0x00000100	Networking events
LOG_LOG	0x00000200	Logging system events
LOG_LOCAL0	0x00001000	Reserved
LOG_LOCAL1	0x00002000	Reserved
LOG_LOCAL2	0x00004000	Reserved
LOG_LOCAL3	0x00008000	Reserved
LOG_LOCAL4	0x00010000	Reserved
LOG_LOCAL5	0x00020000	Reserved
LOG_LOCAL6	0x00040000	Reserved
LOG_LOCAL7	0x00080000	Reserved

Additionally, the log_facility_set_t type is used during query operations. The log_facility_set_t type is defined as an unsigned int.

Severity Type (log_serverity_t)

The `log_severity_t` type defines the severity of the event. The `log_severity_t` type is defined as an unsigned `int`.

Table 10-4: log_severity_t Values

Name	Bit Address	Description
`LOG_EMERG`	0x00000001	Emergency condition
`LOG_ALERT`	0x00000002	Condition requiring immediate correction
`LOG_CRIT`	0x00000004	Critical condition
`LOG_ERR`	0x00000008	Error
`LOG_WARNING`	0x00000010	Warning message
`LOG_NOTICE`	0x00000020	Condition requiring special handling
`LOG_INFO`	0x00000040	Informational message
`LOG_DEBUG`	0x00000080	Debugging message
`LOG_POSIX_SEV`[1]	0x80000000	Sets POSIX standard rather than default error handling

1. Default POSIX behavior can be added by including `LOG_POSIX_SEV` in a function with the `and` or `or` expression. The Default POSIX behavior is to include the highest severity level found, and every severity level beneath.

Log Record Identifier (log_recid_t)

The `log_recid_t` type is used to identify specific records to any log-reading code. Defined as an unsigned `int`.

Event Record Identifier (log_event_t)

`log_event_t` indicates the format of the Event Data portion of the event data. Defined as an unsigned `int`.

Table 10-5: log_event_t Values

Name	Bit Address	Description
`LOG_STRING`		Event Data is a NULL-terminated string.
`LOG_BYTE`		Event Data is a series of bytes.
`LOG_SHORT`	2 bytes each	Event Data is a series of shorts.
`LOG_INT`	4 bytes each	Event Data is a series of ints.
`LOG_USERBASE`	10,000	Minimum user defined type
`LOG_USERMAX`	`INT_MAX`	Maximum user defined type

Log Record (log_entry)

The `log_entry` struct reads the back of the constant part of a log record. `log_entry` contains the following members.

Table 10-6: log_entry Members

Member	Type	Description
`log_status`	`byte`	Status of the record: • bit 0: Available=0, Used=1 • bit 1: Endian of this entry. 1=big, 0=little • bits 2-7: reserved
`log_recid`	`log_recid_t`	System assigned ID of the log record; starts at 1, and increases to a maximum of 65535. 0 represents the header or a daemon error.
`log_size`	`size_t`	Size of the log record's variable data portion
`log_event_id`	`log_event_t`	Log record identification code; provides the format of the data.
`log_facility`	`log_facility_t`	Log record facility code
`log_severity`	`log_severity_t`	Log record severity code

Table 10-6: log_entry Members (Continued)

Member	Type	Description
log_uid	uid_t	Effective User ID associated with event
log_gid	gid_t	Effective Group ID associated with event
log_pid	pid_t	Process ID associated with event
log_pgrp	pid_t	Process group associated with event
log_tid	tid_t	Thread ID associated with event
log_time	struct timespace	System time when event was logged

Log Query (log_query_t)

The log_query_t type is used to filter events in a log.

Table 10-7: log_query_t

Member	Type	Description
q_facility_set	log_facility_set_t	Stores facilities for query.
q_severity	log_severity_t	Stores severity for query.

Writing Log Entries

The log_write() function is used to write information to the System Log. For more information on the process flow, see “Write Flow” on page 152.

log_write()

```
#include <eventlog.h>

int log_write(log_facility_t facility,
int event_id, log_severity_t severity,
const void *buf, size_t len);
```

Description

The log_write() function creates the facility, severity, and data for an event log. After filling the structure, log_write() passes the data to the Event Logging Driver. If the Event Logging Driver is not running, the log_write() function fails.

NOTE: Success of this function indicates that the event is buffered to memory, not written to the Active System Log file.

In addition to the facility set, severity, and size arguments listed in the prototype, `log_write()` also adds arguments defined in the table "log_entry Members" on page 161.

This function is available as both a kernel space and user library function.

Returns

The `log_write()` function returns 0 after a successful completion. If an error occurred, `log_write()` returns an error number.

Table 10-8: log_write() Error Returns

Error	Description
`EINVAL`	The `severity` argument is invalid. `buf` is invalid.
`ENOSPC`	There are no free buffers in the Log System Buffer.
`EMSGSIZE`	The `len` argument exceeds `LOG_ENTRY_MAXLEN`.
`EPERM`	The caller does not have appropriate privileges.
`EIO`	An I/O error occurred when writing to the System Buffer.
`ENODEV`	The Event Log Driver is not running.

Processing Log Entries

The functions `log_open()`, `log_read()`, `log_notify()`, `log_close()`, `log_seek()`, `log_severity_compare()`, `log_facility*()`, and `log_query()` are implemented according to POSIX 1003.1h specifications. These log processing functions are provided only as user library functions.

log_open()

```
#include <eventlog.h>
```

```
int log_open(logd_t *logdes, const char *path,
const log_query_t *query);
```

Description

The `log_open()` function establishes a connection between a log file and a log descriptor. If the `path` parameter is `NULL`, the current Active System Log is used. Otherwise, the `log_open()` function searches the Archived System Log pointed to by `path`. After successful completion, `log_open()` returns a log file handle (in the `logdes` parameter) to the calling process.

NOTE: The log file handle returned by `log_open()` can only be used by the Event Logging System.

Returns

If successful, `log_open()` returns the value 0. If an error occurs, `log_open()` returns an error number.

Table 10-9: log_open() Error Returns

Error	Description
EACCES	Search or read permission is denied on a component of the `path` prefix.
EINVAL	The `query` argument is not `NULL` or does not point to a valid `log_query_t`.
EINVAL	The `path` argument is not a valid Event Log file.
EMFILE	The calling process has too many log descriptors open (`LOG_OPEN_MAX`).
EXDEV	No Active System Log
ENAMETOOLONG	The length of the `path` argument exceeds `PATH_MAX` (80 characters).
EMFILE	Too many files are open in the system.
ENOENT	The file specified by `path` does not exist.
ENOTDIR	A component of the `path` prefix is not a directory.
ENOEXEC	Read of header file or Log file failed.
EBADMSG	Incompatible log file version

log_read()

```
#include <eventlog.h>

int log_read(logd_t logdes,
struct log_entry *entry,void *log_buf,
size_t log_len);
```

Description

The `log_read()` function reads from the file associated with `logdes` file descriptor and matches the current query with an event record from an Active or Archived System Log.

If an event record that matches the log file query is found, it returns the event record and event data to the calling process. If the end of file is reached on the Active System Log before a match is found, an `EAGAIN` error is returned. `EAGAIN` errors are returned on Active System Log files only. When an end of file is reached on an Archived System Log, an `EOF` error is returned.

If the Active System Log is full and there is no match, an `EOF` error is returned. This error occurs only when a user thread is accessing an event log that was closed by the Event Logging Daemon. Use the `log_close()` function to close this file handle. Use `log_open()` with a path of `NULL to` open the new Active System Log.

This is a blocking call and does not wait for a log entry to be written, but rather returns immediately with the result.

Return

If successful, `log_read()` returns the value 0. If an error occurs, `log_read()` returns an error number.

Table 10-10: log_read() Error Returns

Error	Description
EBADF	The `logdes` argument is not a valid file descriptor.
EAGAIN	No data is available: There are no unread matching events remaining and the maximum file size is not reached.
EINTR	A signal has interrupted the call.
EIO	An I/O error has occurred while reading the file.
EOF	End of file is reached without finding a matching event.

Table 10-10: log_read() Error Returns (Continued)

Error	Description
EINVAL	log_buf and/or entry are NULL.
EMSGSIZE	Event record data is larger than log_len.

log_notify()

```
#include <eventlog.h>

int log_notify(logd_t logdes,
const struct sigevent *notification);
```

Description

The function log_notify() allows a calling process to be notified of specific event records. The Event Logging Daemon returns an event record log that matches the logdes file descriptor.

If notification is NULL, no notification is performed. If the calling process has already registered a notification with the same logdes, the new notification replaces the existing one.

The sig_num member of the notification struct must be a real-time signal.

The notification request is not removed when logdes closes. Though this differs from the POSIX specification, it allows notification to continue after the Active System Log is rolled.

NOTE: Although open file handles are inherited by child processes, notifications are not.

If the Event Logging Daemon cannot add the notification to the notification table, the Daemon returns a siginfo_t signal to the calling thread. The si_value of siginfo_t is -1 to signal the thread that an error occurred.

To notify a thread of a match with query, the siginfo_t signal is sent to the thread with a si_value of 0.

Return

On successful completion, `log_notify()` returns 0. If an error occurs, the error number is returned.

Table 10-11: log_notify() Error Returns

Error	Description
EBADF	The `logdes` argument is not a valid file descriptor.
EINVAL	The `notification` argument is invalid.
EPERM	The process requested notification on a log that is not written to.
EBADF	The Log Daemon message queue does not exist.
EBUSY	The notification table in the Log Daemon is full.
EXDEV	Cannot access Daemon
EAGAIN	Daemon busy

log_close()

```
#include <eventlog.h>

int log_close(logd_t logdes);
```

Description

The `log_close()` function deallocates the open file descriptor associated with logdes. This function also performs a `close()` on the file handle.

Return

On successful completion, `log_close()` returns 0. If an error occurs, the error number is returned.

Table 10-12: log_close() Error Returns

Error	Description
EBADF	The `logdes` argument is not a valid file descriptor.

log_seek()

```
#include <eventlog.h>
int log_seek(logd_t logdes,
log_recid_t log_recid);
```

Description

The `log_seek()` function sets the file offset of the open log associated with `logdes` to the event record that matches `log_recid`. Two values other than a record ID can be passed. `LOG_SEEK_START` (defined as 0) and `LOG_SEEK_END` (defined as -1) reposition the start or end of the current log file, respectively.

Return

On successful completion, `log_seek()` returns 0. If an error occurs, the error number is returned.

Table 10-13: log_seek() Error Returns

Error	Description
EBADF	The `logdes` argument is not a valid file descriptor.
ENOENT	No log record exists in the specified log file with a record ID matching `log_recid`.

log_severity_compare()

```
#include <eventlog.h>
int log_severity_compare(int *order,
log_severity_t s1, log_severity_t s2);
```

Description

The `log_severity_compare()` function compares the severity levels of `s1` and `s2`. If `s1` is more severe than `s2`, `order` is set to 1. If `s2` is more severe, `order` is set to -1. If `s1` equals `s2`, `order` is set to 0.

Return

On successful completion, `log_severity_compare()` returns 0. If an error occurs, the error number is returned.

Table 10-14: log_severity_compare() Error Returns

Error	Description
EINVAL	`s1`, `s2`, or both, are invalid severities.

Setting Log Facilities

log_facilityemptyset()

```
#include <eventlog.h>

int log_facilityemptyset(
log_facility_set_t *set);
```

Description

The `log_facilityemptyset()` function excludes all facilities in the facility set pointed to by `set`.

Return

On successful completion, `log_facilityemptyset()` returns 0. If an error occurs, the error number is returned.

Table 10-15: log_facilityemptyset() Error Returns

Error	Description
EINVAL	`set` points to `NULL`.

log_facilityfillset()

```
#include <eventlog.h>

int log_facilityfillset(
log_facility_set_t *set);
```

Description

The `log_facilityfillset()` function includes all facilities in the facility set pointed to by `set`.

Return

On successful completion, `log_facilityfillset()` returns 0. If an error occurs, the error number is returned.

Table 10-16: log_facilityfillset() Error Returns

Error	Description
EINVAL	`set` points to `NULL`.

log_facilityaddset()

```
#include <eventlog.h>

int log_facilityaddset(log_facility_set_t *set,
log_facility_t facilityno);
```

Description

The `log_facilityaddset()` function adds the facility `facilityno` to the facility set to pointed to by `set`.

Return

On successful completion, `log_facilityaddset()` returns 0. If an error occurs, the error number is returned.

Table 10-17: log_facilityaddset() Error Returns

Error	Description
EINVAL	`set` points to `NULL`.

log_facilitydelset()

```
#include <eventlog.h>

int log_facilitydelset(log_facility_set_t *set,
log_facility_t facilityno);
```

Description

The `log_facilitydelset()` function removes the facility `facilityno` from the facility set pointed to by `set`.

Return

On successful completion, `log_facilitydelset()` returns 0. If an error occurs, the error number is returned.

Table 10-18: log_facilitydelset() Error Returns

Error	Description
EINVAL	`set` points to `NULL`.

log_facilityismember()

```
#include <eventlog.h>

int log_facilityismember(
log_facility_set_t *set,
log_facility_t facilityno, int *member);
```

Description

The `log_facilityismember()` function checks if `facilityno` is included in the facility set pointed to by `set`.

Return

On successful completion, `log_facilityismember()` returns 0. If an error occurs, the error number is returned.

Table 10-19: log_facilityismember() Error Returns

Error	Description
EINVAL	`set` points to `NULL`.
EINVAL	`member` is `NULL`.

Querying Log Entries

log_query_init()

```
#include <eventlog.h>

int log_query_init(lot_query_t *query);
```

Description

The `log_query_init()` function initializes the `log_query_t` struct `query`. The default value for `query` is an empty facility set and an empty severity. The `log_query_init()` function does not allocate memory for `query`.

Return

On successful completion, `log_query_init()` returns 0. If an error occurs, the error number is returned.

Table 10-20: log_query_init() Error Returns

Error	Description
`EINVAL`	`query` points to `NULL`.

log_query_destroy()

```
#include <eventlog.h>

int log_query_destroy(lot_query_t *query);
```

Description

The `log_query_destroy()` function is included as a part of the POSIX specification. Because the `log_query_init()` function does not allocate memory for `query`, there is no need to destroy it.

Return

`log_query_destroy()` always returns 0.

log_query_getfacilities()

```
#include <eventlog.h>
int log_query_getfacilities(log_query_t *query,
log_facility_set_t *facility);
```

Description

The `log_query_getfacilities()` function returns the facility for the current `query`.

Return

On successful completion, `log_query_getfacilities()` returns 0. If an error occurs, the error number is returned.

Table 10-21: log_query_getfacilities() Error Returns

Error	Description
`EINVAL`	`query` points to `NULL`.

log_query_setfacilities()

```
#include <eventlog.h>
int log_query_setfacilities(log_query_t *query,
log_facility_set_t *facility);
```

Description

The `log_query_setfacilities()` function sets the current facility set for `query` to the facility set pointed to by `facility`.

Return

On successful completion, `log_query_setfacilities()` returns 0. If an error occurs, the error number is returned.

Table 10-22: log_query_setfacilities() Error Returns

Error	Description
`EINVAL`	`query` points to `NULL`.

log_query_getseverity()

```
#include <eventlog.h>
int log_query_getseverity(log_query_t *query,
log_severity_t *severity);
```

Description

The `log_query_getseverity()` function returns the severities for `query`. The severities are placed into the severity set pointed to by `severity`. The `log_query_getseverity()` function sets the value of `severity` equal to the severity member of `query`.

Return

On successful completion, `log_query_getseverity()` returns 0. If an error occurs, the error number is returned.

Table 10-23: log_query_getseverity() Error Returns

Error	Description
`EINVAL`	`query` points to `NULL`.
`EINVAL`	`severity` points to `NULL`.

log_query_setseverity()

```
#include <eventlog.h>
int log_query_setseverity(log_query_t *query,
log_severity_t *severity);
```

Description

The `log_query_setseverity()` function sets the current severities in `query`.

Return

On successful completion, `log_query_setseverity()` returns 0. If an error occurs, the error number is returned.

Table 10-24: log_query_setseverity() Error Returns

Error	Description
EINVAL	`query` points to `NULL`.
EINVAL	`severity` is an invalid severity.

Sample Driver Code

The following section provides sample code used with the Event Logging System.

Kernel Code Example

```
#include <eventlog.h>
...
int rval;
...
rval = log_write(LOG_KERN, LOG_STRING, LOG_CRIT,
"Kernel error", 12);
...
```

User Code Example

```
#include <sys/types.h>
#include <stdio.h>
#include <signal.h>
#include <ctype.h>
#include <eventlog.h>
logd_t logfd;
int sig_handler(int sig, siginfo_t *extra, void *v)
{log_entry_t e;
char buff[500];
int rc;
     if ((rc = log_read(logfd, &e, buff, 500)) != LOG_OK)
     {
          printf("\nError reading log. rc = %u",rc);
     }
     else
     {
          printf("\nEvent Read: fac %u sev %u data %s",
               e.log_facility, e.log_severity, buff);
          fflush(stdout);
     }
```

```
}

/*****************************************
*                 MAIN
*****************************************/
int main(int argc, char **argv)
{
struct sigaction sa;
int rc;log_query_t q;

struct sigevent not;

     /* set up signal handler for notify */
     sa.sa_handler = (void(*)(_MATCH_ALL))sig_handler;
     sigemptyset(&sa.sa_mask);/* clear signal mask */
     sa.sa_flags = 0;
     if (sigaction(SIGRTMIN, &sa, NULL))
     {
          printf("\nError setting up signal handler.");
          exit(1);
     }

     /* setup the query */
     q.q_facility_set = LOG_EVLOG;
     q.q_severity = LOG_WARNING;

     /* open the active system log */
     if ((rc = log_open(&logfd, NULL, &q)) != LOG_OK)
     {
          printf("\nError opening active system log.");
          exit(1);
     }

     /* set up a notify*/
     not.sigev_signo = SIGRTMIN;

     /* attempt to do a notify on archive...it should fail */
     if ((rc = log_notify(logfd, &not)) != LOG_OK)
     {
          printf("\nError setting up notify.");
          exit(1);
     }

     while(1)
          sleep(1);
     exit(0);
```

APPENDIX A *Glossary*

API - Application Programming Interface - A language interface used by programs to access the operating system and/or applications.

Assembler - A program that converts assembly language into machine-executable code.

Assembly language - A human-readable version of machine code or instructions executed by a computer or microprocessor - A compiler produces assembly language, and an assembler converts it into machine code for execution.

ATM - Asynchronous Transfer Mode - A family of communications protocols.

Binary semaphore - A type of semaphore that only has two states, also called a *Mutex* - Also see *Counting semaphore.*

Board Support Package - See *BSP.*

Boot - The sequence of events, from system power up to starting an operating system and/or application on a system - In order to boot, a boot loader must retrieve a bootable image (OS or application executable) from disk, EPROM, or over a network.

Breakpoint - Used when debugging code, breakpoints halt execution of a program at a certain point, allowing programmers to trace through their application.

BSP - Board Support Package - A collection of programs and device drivers that allow an operating system or other piece of software to execute on a particular hardware platform.

CDK - See *Cross development kit*.

Compiler - A program or collection of programs that converts source code into either assembly language or executable machine code.

Context switch - The process of suspending a currently running thread, task, or process, and beginning or resuming the execution of another thread - In LynxOS there are at least three kinds of context switches: User threads within a virtual

address space (process), User threads across virtual address spaces (inter-process), and LynxOS kernel threads.

Counting semaphore - A semaphore with several states that can be increased or decrease - Also see *Binary semaphore.*

Critical section - A section of code that must be executed without interruption - If two critical sections run at the same time, it is called a race condition.

Cross development - The process of developing an application or kernel on a host system configuration (Linux or Windows, for example) that is different from the target system configuration where the application is to be deployed.

Cross development environment - The cross development environment includes LynxOS-specific compilers, linkers, libraries, and other development tools. For example, GDB - the LynuxWorks version of the GNU debugger, is provided with LynxOS. The LynxOS cross development environment allows for creating LynxOS applications and kernels from a variety of platforms.

Cross development kit - A suite of LynxOS cross development tools specific to a particular host system configuration - Supported cross development kit platforms include Windows, Linux, and Sun/Solaris.

Device driver - Software that facilitates the interfacing of applications or programs with system hardware.

Debugger - A tool for debugging - Specifically, one that facilitates the controlled execution of a program and the inspection of program data.

Determinism - The attribute of a system displaying known and measurable performance characteristics - The kinds of determinism of interest to real-time developers include maximum blocking times, interrupt latencies, and context switch times.

DMA - Direct Memory Access - Protocol allowing for data transfers between two peripherals (memory or I/O devices) without passing it through the processor.

DRM - Device Resource Manager - LynxOS module that manages buses and devices on a system - The DRM provides a standard set of services and calls used to access buses and devices.

EEPROM - Electrically Erasable Programmable Read-Only Memory - Non-volatile ROM that the user can program, erase and reprogram as needed - EEPROM memory is erased with a small electrical charge.

EPROM - Erasable Programmable Read-Only Memory - Non-volatile ROM that the user can program, erase and reprogram as needed - EPROM memory is erased with exposure to ultraviolet light.

Embedded system - A combination of hardware and software designed to perform a specific function.

Fault tolerance - The characteristic of a system enabling it to recover from hardware and software faults, usually through the use of redundant hardware.

Firmware - Software that has been written to ROM.

Flash memory - EPROM that can be erased and rewritten by software.

Hard real-time - A hard real-time system must always meet specific deadlines. In a soft real-time system, deadlines can be missed. A hard real-time system is used to denote a critical environment where deadlines cannot be missed. See also *Real-time operating system.*

High availability - The utilization of distributed computer resources to maximize system uptime and provide failsafe mechanisms for hardware or software failures

Host - A computer or operating system that communicates with a target system via a serial port or network connection - In cross development environments, the host system is where applications are developed. Also see *Target*.

Hot swap - The act of inserting and extracting hardware boards to/from a system chassis without having to cycle system power.

In-circuit emulator - A hardware device that either substitutes or augments the functioning of a microprocessor - Emulation enables the controlled execution of software programs and the exercising of system hardware. In-circuit emulation is useful when porting LynxOS to new hardware, or creating a BSP.

Inter-process communication - The act and the mechanisms for communication and synchronization among threads, tasks, or processes - IPCs include control variables, mutexes, queues, and semaphores.

Interrupt - A hardware event, such as the arrival of data on an I/O port, that causes a specific asynchronous software response, that is, the program stops its current activity to service the interrupt. The LynxOS model of interrupt processing is to field the interrupt in the device driver, but to perform actual processing in LynxOS kernel threads.

KDI - Kernel Downloadable Image - KDIs combine a LynxOS kernel and associated applications into bootable images designed for easy downloading to target systems.

Kernel - The core of the operating system that handles thread or task creation, scheduling, and synchronization.

Kernel threads - In LynxOS, kernel threads provide processing service to device drivers.

Linker - A utility program that combines the object code output of a compiler or an assembler to create an executable program.

Microcontroller - Similar to microprocessors, microcontrollers are designed specifically for use in embedded systems.

Microkernel - A type of kernel architecture that breaks kernel functionality and services into modules - Users can select only the required kernel modules, reducing the kernel footprint size, and removing unneeded services.

Microprocessor - A silicon chip that contains a general-purpose CPU

MMU - Memory Management Unit - The circuitry present in a microprocessor or sometimes in a separate device to allow for the protection of regions of system memory and for the mapping & translation of logical (program) addresses into physical memory address to implement virtual addressing.

Mutex - A multitasking-aware flag used to protect critical sections from interruptions - Mutexes are also called *Binary semaphores*. Also see *Critical sections*.

Mutual exclusion - Achieved through use of semaphores or mutexes, mutual exclusion allows for exclusive access to a shared resource.

Native development - The process of developing applications on the same platform and environment that the application is to be deployed on - Also see *Cross development*.

Netboot - Booting an application or kernel image across a network instead of from ROM or a disk.

ODE - Open Development Environment - The native LynxOS development environment - An ODE consists of LynxOS native versions of GNU compilers, the LynxOS/GNU linker, libraries, ROMing tools, utilities, and other utilities. Also see *Cross development kit*.

Policy - A set of user-defined behaviors in a system - In real-time, policy usually refers to the rules for scheduling threads of like priority. LynxOS supports three scheduling policies: Round-robin (time-slice), Quantum (N time-slices), and FIFO (run to completion). In high availability systems, policy can also mean the specific responses to failure conditions.

POSIX - Portable Operating System Interface - A family of standards based on UNIX system V and Berkeley UNIX that defines the interface between

applications and an operating system for maximum compatibility and portability across implementations.

Preemption - The interruption of thread execution by the operating system for the purpose of rescheduling - Preemption occurs when a program's time-slice or quantum has expired, or when a higher-priority thread becomes ready to run. In LynxOS, the operating system itself is preemptible.

Priority - The relative importance of a thread of execution - Priorities are represented in numerical values. LynxOS offers 256 unique priorities. In a hard real-time system, the thread with the highest priority that is ready to run always runs.

Process - An executing program - In LynxOS, processes are virtually addressed containers for globally structured threads.

PROM - Programmable Read-Only Memory.

Real-time operating system - An operating system that responds to events and input immediately - General operating systems, such as DOS or UNIX, are not real-time.

RAM - Random Access Memory - Volatile memory type that allows data to be written to and erased to/from it

ROM - Read Only Memory - Non-volatile memory type that only allows read-access.

RTOS - See *Real-time operating system*.

Scheduling - The process of determining which thread or task is allowed to run on a system - Hard real-time scheduling is based on priority. If two threads exist at the same priority, scheduling is then based on policy.

Semaphore - A hardware or software flag that indicates the status of a resource

Target - The system on which an application is deployed - Typically, the target system is synonymous with embedded system. Also see *Host*.

Task - The basic schedulable entity in most operating systems, especially real-time operating systems - Usually synonymous with process.

Thread - Part or parts of a program that can be executed independent of the parent process.

Virtual addressing - The use of an MMU to provide applications the illusion of a logical address space - Each virtual address space is isolated from other applications and their virtual address spaces, except as requested by memory sharing inter-process communications. Virtual addressing translates the program-

local logical addresses into physical addresses, and maps blocks of physical memory into logical address spaces only as needed.

Virtual memory - The extension of physical memory (RAM) onto long-term storage, typically a disk drive, through the use of virtual addressing and swapping.

Volatile - Memory that loses its contents after system power is cycled - The contents of non-volatile memory do not change after system power is cycled.

X Windows - A graphical user interface and windowing environment for UNIX systems.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z