ChorusOS 5.0 Features and Architecture Overview

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806–6897–10
December 2001

Copyright 2001 Sun Microsystems, Inc. 901 San Antonio Road Palo Alto, CA 94303-4900 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, Sun Embedded Workshop, Sun WorkShop, Netra, ChorusOS, Solstice, JDK, Java, J2ME, JVM, Solaris Web Start Wizards and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Adobe is a registered trademark of Adobe Systems, Incorporated. Adobe logo is a trademark or registered trademark of Adobe Systems, Incorporated.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Federal Acquisitions: Commercial Software–Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2001 Sun Microsystems, Inc. 901 San Antonio Road Palo Alto, CA 94303-4900 U.S.A. Tous droits réservés

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, Sun Embedded Workshop, Sun WorkShop, Netra, ChorusOS, Solstice, JDK, Java, J2ME, JVM, Solaris Web Start Wizards et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc. Adobe est une marque enregistreé de Adobe Systems, Incorporated. Adobe logo est une marque de Adobe Systems, Incorporated.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Adobe PostScript

011121@2870

Contents

Tables

Figures

Preface

The *ChorusOS 5.0 Features and Architecture Overview* describes the features and architecture of the ChorusOS operating system. The *ChorusOS 5.0 Features and Architecture Overview* serves as a general overview of the architecture of the ChorusOS operating system, presenting its particularities and advantages, as well as introducing all its features and components.

Who Should Use This Book

The ChorusOS 5.0 Features and Architecture Overview is intended for all users. It provides a global view of the ChorusOS operating system and as such does not present any procedural information or details of implementation.

Before You Read This Book

This book serves as a stand-alone presentation of ChorusOS and as such does not necessarily require you to read other manuals first.

How This Book Is Organized

The ChorusOS 5.0 Features and Architecture Overview is organized as follows:

Chapter 1 provides a general introduction to the ChorusOS operating System.

Chapter 2 describes the architecture and main advantages of the ChorusOS operating system.

Chapter 3 presents the features ChorusOS architecture.

Appendix A lists the optional components that can be configured in into an instance of the ChorusOS operating system.

Appendix B provides a complete list of the system calls available in the ChorusOS operating system.

Glossary provides a glossary of ChorusOS operating system terminology.

Related Books

This book provides a global overview of the ChorusOS operating system. For precise information regarding installation, development, implementation, deployment and administration, please consult the relevant volumes of the rest of the ChorusOS documentation set.

For information about the structure of the ChorusOS documentation set, see *About ChorusOS 5.0 Documentation*.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the ChorusOS 5.0 Operating System

This chapter provides a brief introduction to the ChorusOS operating system, describing its purpose, architecture, the types of target supported, and the enhancements added in version 5.0.

What is the ChorusOS Operating System?

The ChorusOS operating system is a highly scalable and reliable embedded operating system that has established itself among top telecommunications suppliers. The ChorusOS operating system is used in public switches and PBXs, as well as within access networks, cross-connect switches, voice-mail systems, cellular base stations, web-phones, and cellular telephones.

The Sun Embedded Workshop™ software provides a development environment with the necessary tools to build and deploy the ChorusOS operating system on a telecommunications platform. The ChorusOS operating system is the embedded foundation for Sun's Service-Driven Network. Offering high service availability, complete hardware and software integration, management capabilities and Java™ technology support dedicated to telecom needs, the ChorusOS operating system allows the dynamic and cost-efficient deployment of new features and applications while maintaining the reliability and functionality of existing networks.

The ChorusOS operating system supports third-party protocol stacks, legacy applications, and applications based on real-time and Java technology, on a single hardware platform.

Component-Based Architecture

The ChorusOS operating system can be tuned very finely to meet the requirements of a given application or environment. The core executive component is always present in an instance of the ChorusOS operating system. Optional features are implemented as components that can be added to, or removed from, an instance of the ChorusOS operating system.

Each API function in the ChorusOS operating system is contained in one or more of the configurable components. As long as at least one of these components is configured into a given instance of the operating system, the function is available. Some library functions are independent of any specific component and are always available.

The optional ChorusOS operating system components are listed in Appendix A.

Supported Target Families

ChorusOS 5.0 runs over Solaris operating environments, and supports the following targets:

- UltraSPARC II (CP1500 and CP20*x*0)
- Intel *x*86, Pentium
- Motorola PowerPC 750 and 74*x*0 processor family (mpc7*xx*)
- Motorola PowerQUICC I (mpc8*xx*) and PowerQUICC II (mpc8260) microcontrollers

What's New in 5.0?

The following new features have been added in release 5.0 of the ChorusOS operating system, with an emphasis given to enhancing the high availability and real time services:

TABLE 1–1 New Features in ChorusOS 5.0

Feature	Description
Black Box	To provide enhanced tracking of system failures.
IPv6	IPv6 base services and commands.
NTP	Network Time Protocol
POSIX real-time API	Including Portable Operating System Interface (POSIX) signals, real-time signals, and process management to complement the existing ChorusOS API, providing a standard, easy migration of UNIX code and bringing the ChorusOS operating system closer to the Solaris operating environment.
Shared Libraries	To extend the existing dynamic libraries feature.
System Events	To notify user-level applications of events in the system or the drivers.
System Instrumentation	System resource instrumentation (with counters and gauges) to control resource usage and anticipate possible over-use or starvation.
System Logging	To provide further logging facilities.
Watchdog Timer Protection	To monitor the system and take action in case of failure.

These new features are covered in more detail in Chapter 2 and Chapter 3 in this overview.

CHAPTER 2

Architecture and Benefits of the ChorusOS Operating System

This chapter describes the general architecture and the benefits provided by the ChorusOS operating system.

General Architecture

The architecture of the ChorusOS operating system is divided into layers built one on top of the other, as illustrated in the following figure.

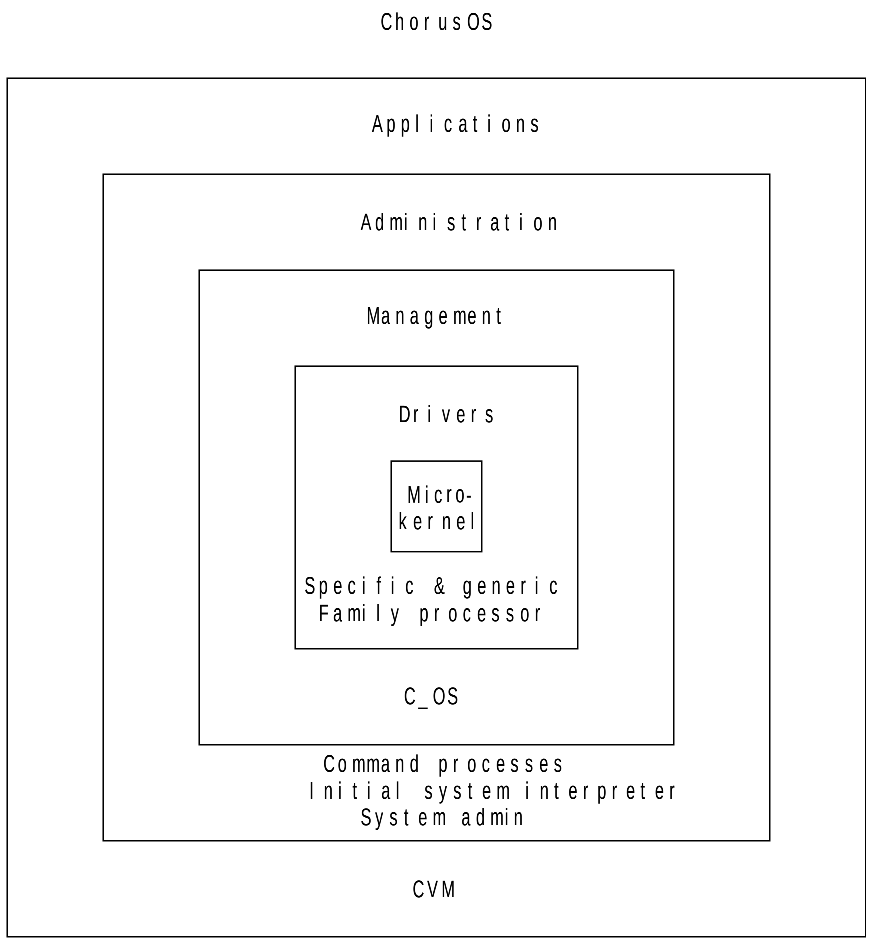

FIGURE 2–1 The Layered Architecture of the ChorusOS Operating System

The following sections provide descriptions of the key components and main advantages of the ChorusOS operating system.

Multi-Platform Development Environment

The Sun Embedded Workshop software, the ChorusOS operating system development environment, provides the tools and libraries for developing C and C++ applications on a range of supported platforms. Development takes place on one system, the host (UltraSPARC), and the operating system is deployed on one or more supported reference target boards.

The ChorusOS operating system also provides several utilities for managing the operating system and applications running on the target. These utilities include components that can be added to the operating system configuration.

The development environment includes:

C & C++ Development Toolchain: GNU gcc and g++ cross-compilers.

C & C++ Symbolic Debugger: GNU GDB debugger for the ChorusOS operating system allows you to see what is going on inside an application while it executes or what an application was doing at the moment it crashed. The GDB Debugger offers the following features:

- Easy-to-use graphical user interface (GUI)
- Support for debugging several applications running on multiple targets with different processor architectures
- Debugging of multithreaded user and supervisor applications, including relocatable applications
- Flexible thread handling: one window per thread, breakpoint per thread or per application
- Visualization of ChorusOS abstractions related to debugged applications or global to the system
- Application debug over Ethernet or serial line, and system debug over serial line
- Ability to debug almost every piece of code, including the boot process, the C_OS, the drivers and the applications

An Embedded Debugger: Which provides symbolic debugging for all system applications and can be called automatically in the case of unrecoverable error.

Configuration Tools: The ChorusOS operating system is configured simply by providing a list of the required components. The configuration tools provided, as part of the Sun Embedded Workshop software, manage any hidden dependencies or possible incompatibilities.

The configuration tools are designed to be flexible enough to be extended to configure any other system component (OS or drivers) or even application actors that are part of the ChorusOS operating system image.

You can use either the graphical interface, called *Ews*, or a command-line interface to view and modify the characteristics of a ChorusOS operating system image. In addition to the possibility of selecting only the required components for the operating system, the Sun Embedded Workshop software supports three other levels of system configuration:

Resources	For the list of components selected, it is possible to fix the amount of resources to be managed, and to set the value of tunable parameters; for example, the amount of memory reserved for network buffers.
Boot Actors	It is possible to include additional actors in the system image loaded at boot time.
Environment	System-wide configuration parameters can be fixed by setting UNIX-like environment strings that the operating system and actors retrieve upon initialization; for example, an IP address can be defined globally by setting `LOCAL_INADDR="192.33.15.18"` using the configuration tool.

A set of Libraries:

- Thread-safe C++
- ANSI-C (POSIX 1003.1 compliant)
- POSIX 1003.1 timers, message queues, shared memory, and semaphores
- POSIX 1003.1 pthreads
- POSIX 1003.1g sockets
- BSD File I/O
- Thread-safe mathematical ANSI-C
- Mathematical IEEE-754
- Library resolve for DNS/NIS client access

- C++ iostream
- C++ exceptions
- C++ STL Management of per-thread private data
- LDAP
- Sun RPC
- X11, Xaw, Xext, Xmu, and Xt libraries

The Sun Embedded Workshop software also provides several utilities for managing the operating system and applications running on the target. These utilities include components that you can add to the operating system configuration.

The application management utilities include:

`Netboot`:	Used to boot the ChorusOS operating system remotely using TFTP, when the target does not provide an embedded boot facility
Default Console:	Used to direct all console I/O to a local display or to a remote host via a serial line
Remote Shell (`rsh`):	Used to execute commands remotely on the target from the host; in particular, this feature allows applications to be loaded dynamically
Resource Status:	Used to list the current status of all operating system resources. For example, actors, threads, and memory.
Logging (`LOG`):	Used to log operating system events as they occur on the target
Monitoring (`MON`):	Used to monitor operating system objects, so that user-defined routines are called when certain operations are performed on specified objects
Profiling:	Used to run profiling sessions on system applications
Benchmarking (`PERF`):	Used to benchmark the operating system

ChorusOS-Solaris Convergence

The ChorusOS operating system is the real-time embedded operating system companion to the Solaris operating environment, providing the following advantages:

Design flexibility
: The sharing between the two operating systems means developers can decide exactly how much of their solution will use the Solaris environment, and how much will be based on the ChorusOS operating system, but use all the same surrounding technology.

Gradual migration

By making it easy for developers to choose any percentage split between the ChorusOS operating system and the Solaris operating system, and by making all the surrounding modules and technologies converge with both, Sun enables developers and their operator clients to begin at any point in the evolution from real-time embedded to computer-based systems, and migrate slowly to where they want to go as they replace obsolete systems and invest in new technologies.

Portable Binary System

For each supported target processor family, the ChorusOS operating system comes with the implementation of at least one reference target board and provides a complete set of well-defined interfaces allowing you to port the ChorusOS operating system to other reference boards in the same target family. The Boot Kernel Interface (BKI) and Device Driver Interface (DDI) available in the binary release of ChorusOS allow you to customize the boot method and to add new drivers.

Configurability

The ChorusOS operating system uses a flexible, component-based architecture that allows different services to be configured into the runtime instance of the ChorusOS operating system. This allows the runtime instance of the ChorusOS operating system to be finely tuned according to the underlying hardware platform, the memory footprint requirements, and the application features.

Essential services required to support real-time applications running on the target system are provided by the core executive, and each optional feature of the operating system is implemented as a separate runtime component that can be added to, or removed from the operating system, as required. This means that the operating system can be configured very accurately to meet the exact needs of a given application or environment, saving on memory and improving performance. Additional components can be added, to create a tailor-made instance, to the level of the resources available.

The core executive can support multiple, independent applications running in both user and supervisor memory space. The core executive can be complemented with additional components to add the features required to support a given application. Additional components supplied by the Sun Embedded Workshop software include:

- Framework and drivers for building board support packages

- Interrupt management
- Processor scheduling
- Synchronization
- Memory management
- Communications
- Time management
- File systems
- Network protocols
- Dynamic process management
- POSIX application programming interfaces (APIs)
- Support for Java applications

This flexible architecture is shown in Figure 2–2.

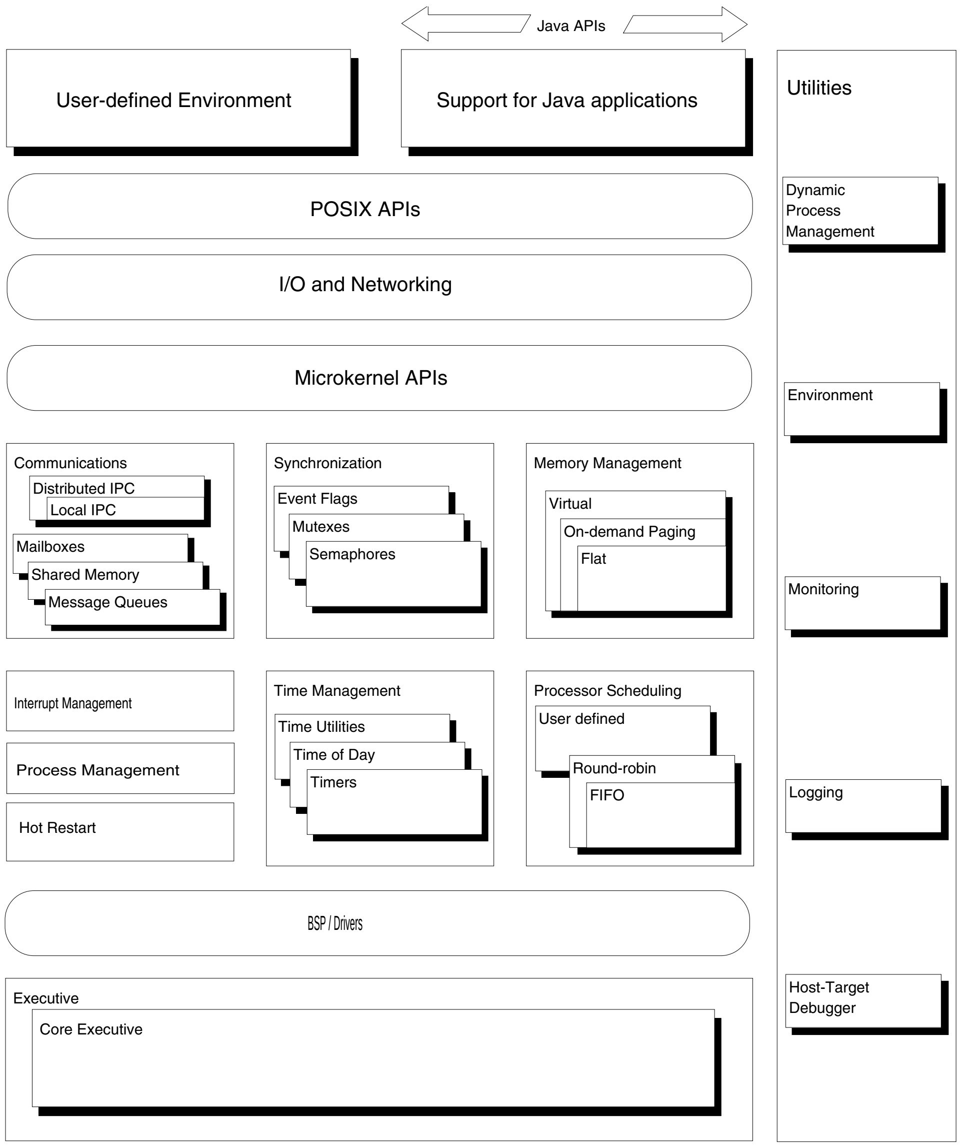

FIGURE 2–2 The ChorusOS Component-based Architecture

Detailed descriptions of the optional features for the ChorusOS operating system are provided in the *ChorusOS 5.0 Application Developer's Guide*.

By taking advantage of the component-based architecture, the application developer can choose between an extremely small operating system that offers simple scheduling and memory options, or a fully-featured, multi-API software platform.

As well as making it possible to produce multiple versions of the operating system, each of which is optimized for its own environment, the component-based architecture provides the following additional benefits:

- Applications developed to run on a minimal configuration can also run unchanged on a more complex configuration, thus providing an evolutionary path for right-sizing devices and systems.
- The programming interfaces for the operating system components are available publicly, providing an open environment for combining third-party system software and development tools.

Configuration Profiles

The ChorusOS operating system provides two standard configuration profiles. These serve as starting points for defining your own configuration:

- Basic profile
- Extended profile

Basic Profile

The basic profile is an example of a small deployment system and defines a realistic configuration while keeping the footprint as small as possible. When using the basic profile, all applications are usually embedded in the system image and launched either at boot time as boot applications, or subsequently from the file system.

Extended Profile

The extended profile is an example of a development system and should be viewed as a reference configuration for telecommunications systems. It includes support for networking using remote Inter-Process Communication (IPC) over Ethernet and a Network File System (NFS) client, using the protected memory model. It allows the development and loading of multi-actor applications. These actors may use any ChorusOS API, provided that the corresponding feature is part of the system configuration.

System Image

Optional features are implemented as components that can be added to, or removed from, an instance of the ChorusOS operating system, known as the system image. The system image is made up of binary or executable files that define the operating system and initial application processes. Once you have built your system image, you are ready to start using the ChorusOS operating system. In this way, the operating system can be very finely tuned to meet the requirements of a given application or environment. The core executive component must always be present in a ChorusOS system image.

Each API function in the ChorusOS operating system is contained in one or more of the configurable components. As long as at least one of these components is configured into a given system image, the function is available to be called. Some library functions are independent of any specific component and are always available.

APIs in the ChorusOS Operating System

The ChorusOS operating system offers three sets of application programming interfaces, including POSIX API compatibility. Each set is intended to serve a certain class of applications.

POSIX Processes

These applications have access to a fully POSIX-compliant API for process management, signals, threads, input/output, memory management, thread sysnchronization, and timing services. This greatly extends the functionality of the ChorusOS operating system, at the same time as allowing easier migration to applications developed by the user, and closer conformity with the Solaris operating environment.

These applications also have access to APIs developed for the ChorusOS services (non existent in POSIX) that conform to the POSIX style, such as system events, black box, watchdog, IPC, and MIPC.

ChorusOS Actors

These include ChorusOS microkernel actors that run the system and the drivers. These applications have access to APIs for all ChorusOS microkernel services.

ChorusOS 4.*x* Legacy applications

An API developed for release 4.x of the ChorusOS operating system, to support applications developed with POSIX-like functionality without being fully POSIX-compliant. Now superseded by the full POSIX implementation described above, but retained for reasons of backward compatibility.

Caution – The POSIX processes and the ChorusOS actors combine to provide a rich range of functionality. However, the two APIs are mutually exclusive. For details of the implementations of these APIs in release 5.0 of the ChorusOS operating system, see `API`(5FEA).

Microkernel

The microkernel is the heart of the ChorusOS operating system and contains the minimum elements required to make a functioning system. In addition to the optional components you can configure into your ChorusOS operating system, the microkernel contains the `kern`, private data manager (`pd`), persistent memory manager (`pmm`), and core executive components. The `kern`, `pmm`, and `pd` provide a minimum set of interfaces that are used by the remainder of the operating system:

- The `kern` must be included in your system image. It implements the microkernel interface and contains the `KERN` actor, the `mk` library and associated header files.
- The `pd` implements the per-thread data interface between the microkernel subsystems, such as the UNIX subsystem.
- The `pmm` implements the persistent memory interface. The `pmm` is included automatically in the system image when the `HOT-RESTART` feature is activated (see "Hot Restart and Persistent Memory" on page 82).

The services provided by the core executive are explained below.

Core Executive

The essential services required to support real-time applications are provided by the core executive. The core executive can support multiple, multithreaded applications running in both user and supervisor memory space.

The core executive implements the basic ChorusOS execution model and provides the framework for all other configurable features. Every system image must include the core executive.

The core executive provides the following functionality:

- Support for multiple, independent applications
- Support for user and system (trusted) applications
- The unit of application modularization (actor)
- The unit of execution (thread)
- Thread control operations
- Local Access Point (LAP) management
- Exception management services
- A minimal interrupt management service

No synchronization, scheduling, time, communication or memory management policies are wired into the core executive. These policies and services are provided by additional features, that the user selects depending on the particular hardware and software requirements.

Actors

This section provides an introduction to actors in the ChorusOS operating system. For further information regarding topics such as loading actors, spawning actors, and their execution environment and communications, see the *ChorusOS 5.0 Application Developer's Guide*.

Actor Definition

An actor is the unit of loading for an application. It serves also as the encapsulation unit to associate all system resources used by the application and the threads running within the actor. Threads, memory regions and communication end points are some examples of these resources. These are covered in detail in the *ChorusOS 5.0 Application Developer's Guide*. All system resources used by an actor are freed upon actor termination.

Some resources, known as anonymous resources, are not bound to a given actor. These must be freed explicitly when they are no longer required. Examples of anonymous resources are physical memory, reserved ranges of virtual memory, and interrupt vectors.

The ChorusOS operating system is dedicated to the development and execution of applications in a host-target environment where applications are developed, compiled, linked, and stored on a host system and then executed on a reference target board where the ChorusOS operating system is running. When configured correctly, the ChorusOS operating system offers convenient support for writing and running distributed applications.

Within the ChorusOS operating system environment, an application is a program or a set of programs, usually written in C or C++. In order to run, an application must be loaded on the ChorusOS runtime system. The normal unit of loading is called an actor and is loaded from a binary file located on the host machine. As with any program written in C or C++, an actor has a standard entry point:

```
int main()
 {
 /* A rather familiar starting point, isn't it? */
 }
```

The code of this type of application will be executed by a main thread that is created automatically by the system at load time. The ChorusOS operating system provides means to create and run more than one thread dynamically in an actor. It also offers services that enable these actors, whether single-threaded or multi-threaded, to cooperate, synchronize, locally or remotely exchange data, or get control of hardware events, for example. These topics are covered step-by-step in the *ChorusOS 5.0 Application Developer's Guide*.

An actor can be of two types: either a supervisor actor or a user actor. These types define the nature of the actor address space. User actors have separate and protected address spaces so that they cannot overwrite each other's address spaces. Supervisor actors use a common but partitioned address space. Depending on the underlying hardware, a supervisor actor can execute privileged hardware instructions, such as initiating an I/O, while a user actor cannot. See "User and Supervisor Actors" on page 34.

Note – In flat memory, supervisor and user actors share the same address space and there is no address protection mechanism.

Binary files from which actors are loaded can also be of two types: either absolute or relocatable. An absolute binary is a binary where all addresses have been resolved and computed from a well-known and fixed basis that cannot be changed. A relocatable file is a binary that can be loaded or relocated at any address.

Both user and supervisor actors can be loaded either from absolute or relocatable binary files. However, common practice is to load them from relocatable files to avoid a static partitioning of the common supervisor address space, and to allow the loading of user actors into this space in the flat memory model. This is covered in detail in "User and Supervisor Actors" on page 34.

Naming Actors

Every actor, whether it is a boot actor or a dynamically-loaded actor, is uniquely identified by an actor capability. When several ChorusOS operating systems are cooperating together over a network in a distributed system, these capabilities are always unique through space and time. An actor may identify itself with a predefined capability, for example:

`K_MYACTOR`.

In addition, an actor created from the POSIX personality is identified by a local process identifier.

```
host% rsh target hello
Started pid = 13
host%
```

Where *target* is the name of your target.

User and Supervisor Actors

There are two main kinds of actor run within the ChorusOS operating system environment: *user* actors and *supervisor* actors.

A user actor runs in its own private address space so that if it attempts to reference a memory address that is not valid in its address space, it encounters a fault and, by default, is automatically deleted by the ChorusOS operating system.

Supervisor actors do not have their own fully-contained private address space. Instead, they share a common supervisor address space, which means that an ill-behaved supervisor actor can access, and potentially corrupt, memory belonging to another supervisor actor. The common supervisor address space is partitioned between the ChorusOS operating system components and all supervisor actors.

As supervisor actors reside in the same address space, there is no memory context switch to perform when execution switches from one supervisor actor to another. Thus, supervisor actors provide a trade-off between protection and performance. Moreover, they allow execution of privileged hardware instructions and so enable device drivers, for example, to be loaded and run as supervisor actors.

On most platforms, the address space is split into two ranges: one reserved for user actors and one for supervisor actors (see Figure 2–3). As user actor address spaces are independent and overlap each other, the address where these actors run is usually the same, even if the actors are loaded from relocatable binaries. On the other hand, available address ranges in supervisor address space may vary depending on how many and which supervisor actors are currently running. Since the ChorusOS operating system is able to find a slot dynamically within the supervisor address space to load an actor, the user does not need to be aware of the partitioning of the supervisor address space: using relocatable binary files is sufficient.

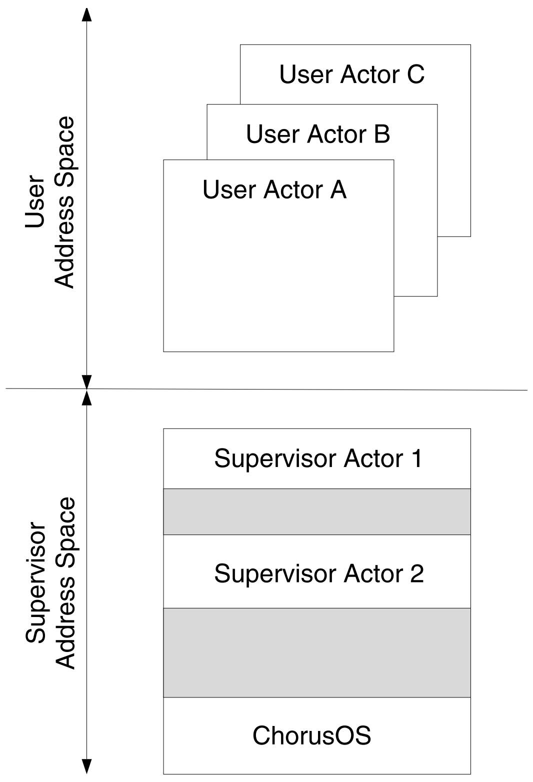

FIGURE 2–3 User and Supervisor Address Spaces

In addition to being either a user or supervisor actor, an actor can be *trusted*, which gives it the right to call certain privileged system services. Trusted actors are also referred to as system actors. A supervisor actor is by definition trusted.

Inter-Actor Communication

The ChorusOS operating system offers a set of services for communicating between actors. Two actors can be made to communicate by sharing memory. Other communication mechanisms can be split into two categories:

- Mechanisms that are local, that is, they do not allow actors running on different machines to communicate. The shared memory mechanism is one of these. You can use the system features in order to implement distributed shared memory. Message queues and local access points are other local communication mechanisms.
- Mechanisms that can be used transparently in a distributed way. The IPC service enables actors to exchange messages transparently whether they are running on the same machine or not.

Threads

This section provides an overview of the use of threads in the ChorusOS operating system. For further information regarding threads, thread handling, thread synchronization, thread scheduling, per-thread data, and threads and libraries, see the *ChorusOS 5.0 Application Developer's Guide*.

Within an actor, whether user or supervisor, one or more threads may execute concurrently. A thread is the unit of execution in a ChorusOS operating system and represents a single flow of sequential execution of a program. A thread is characterized by a context corresponding to the state of the processor (registers, program counter, stack pointer or privilege level, for example). See Figure 2–4.

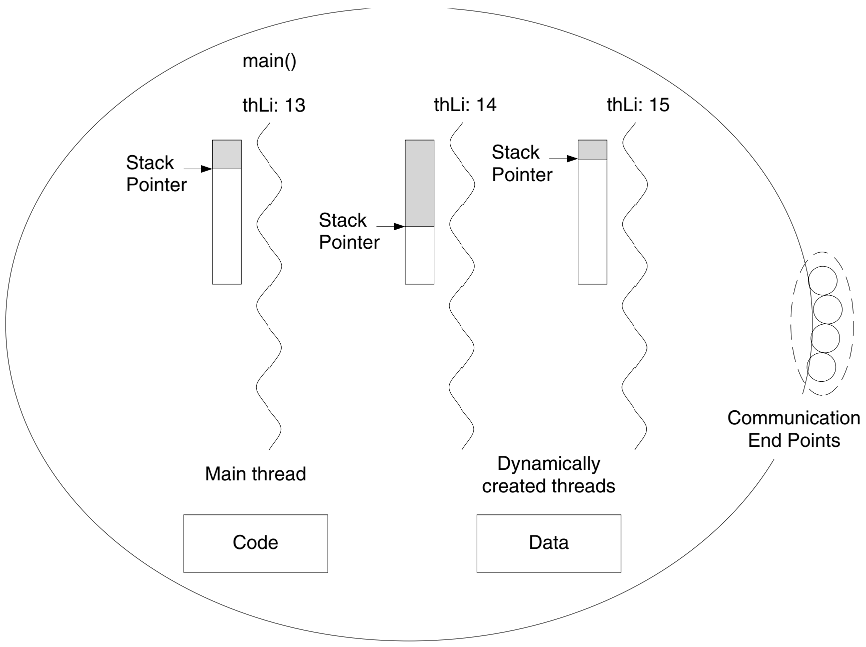

FIGURE 2–4 A Multi-Threaded Actor

Threads can be created and deleted dynamically. A thread may be created in an actor other than the one to which the creator thread belongs, provided they are both running on the same machine. The actor in which the thread was created is called the home actor or the owning actor. The home actor of a thread is constant during the life of the thread.

The system assigns decreasing priorities to boot actor threads, so that boot actor main threads are started in the order in which they were loaded into the system image. If a boot actor's main thread sleeps or is blocked, the next boot actor threads will be scheduled for running.

Although there are no relationships maintained by the ChorusOS operating system between the creator thread and the created thread, the creator thread is commonly called the parent thread, and the created thread is commonly called the child thread.

A thread is named by a local identifier referred to as a thread identifier. The scope of this type of identifier is the home actor. In order to name a thread of another actor, you must provide the actor capability and the thread identifier. It is possible for a thread to refer to itself by using the predefined constant: `K_MYSELF`.

All threads belonging to the same home actor share all the resources of that actor. In particular, they may access its memory regions, such as the code and data regions, freely. In order to facilitate this access, the ChorusOS operating system provides synchronization tools, covered in "Synchronization" on page 53 in this book.

Threads are scheduled by the microkernel as independent entities The scheduling policy used depends on the scheduling module configured within the system. In a first approach, assume that a thread may be either active or waiting. A waiting thread is blocked until the arrival of an event. An active thread may be running or ready to run.

POSIX Services

The ChorusOS operating system offers a POSIX API for process management. POSIX comprises a set of standard APIs for portable multithreaded programming. The ChorusOS operating system provides the following POSIX APIs:

- POSIX (1003.1) message queue (`POSIX-MQ`)
- POSIX (1003.1) semaphores (`POSIX-SEM`)
- POSIX (1003.1) shared memory objects (`POSIX-SHM`)
- POSIX (1003.1) real-time clock/timers (`POSIX-TIMERS`)
- POSIX (1003.1) pthreads (`POSIX-THREADS`)
- POSIX (1003.1) compatible I/O system calls (`POSIX-FILEIO`)
- POSIX (1003.1) compatible socket system calls (`POSIX-SOCKETS`)
- POSIX (1003.1) real-time signals (`POSIX_REALTIME_SIGNALS`)

For a specific POSIX-compatible function to be available, the component in which it is contained must be configured into the operating system. In some cases, a function can be contained in more than one component, therefore, at least one of the components must be selected.

The processes also benefit from the dynamic libraries (`DYNAMIC_LIB`) and compressed (`GZ_FILE`) features. Processes can be multi-threaded using the POSIX `pthread` calls described below. However, the ChorusOS operating system has some limitations regarding multi-threaded processes. It is not possible to invoke either `fork()` or `exec()` from a multi-threaded process. Such attempts will fail and report an error code. If a multi-threaded application needs to launch a process, it should use the `posix_spawn()` system call. However, the ChorusOS implementation of the `posix_spawn()` call is limited and does not permit handling of file or signal management operations.

Process images in the ChorusOS operating system are loaded from their files and are not mapped in memory, even though the underlying selected memory profile supports paging.

You can set tunable values for the following when you build your system:

- The maximum number of processes that can be created on a system. This value is lower than the maximum number of actors that can be created, which is also set when you build your system.
- The maximum number of threads that can be created on a system.
- The maximum number of threads that can be created inside a process.

User and Supervisor Processes

POSIX processes are divided into two types, *user* and *supervisor* processes. The main differences between user and supervisor processes are outlined below:

- Supervisor processes run in the same address space as the system. User processes run in their own private address space so that if they attempt to reference a memory address that is not valid in their address space, they encounter a fault and, by default, are deleted.
- Supervisor processes run in privileged mode. This is architecture-specific, and means that certain privileged instructions are available to supervisor processes which are not available to user-mode applications.
- User applications must trap into the system, whereas supervisor applications do not since they operate in privileged mode.
- Supervisor applications have only one stack (the system stack), whereas user applications have one stack in user mode and another stack to execute when they trap into privileged mode.
- Some libraries are different between supervisor applications and user applications, since supervisor applications can execute privileged instructions, and user applications trap to access system services.

High Availability

The ChorusOS operating system incorporates several features that provide high availability services, including:

- Black Box
- Dynamic Reconfiguration
- Memory Protection
- Watchdog timer

Black Box

The black box feature provides an enhanced mechanism for tracing the activity of the system, so that the exact cause of any failure can be determined quickly and easily.

Black box timer is elaborated further in "Black Box (`BLACKBOX`)" on page 124.

Dynamic Reconfiguration

The dynamic process management feature of the ChorusOS operating system allows processes to be loaded dynamically, from either disk or the network, without first halting the system. This provides the basis for a dynamic reconfiguration capability that minimizes service downtime, and keeps existing services available while the system is modified or upgraded. Dynamic reconfiguration also relies on the IPC facilities of the ChorusOS operating system to transfer inbound communication to the new processes transparently.

For example, with the ChorusOS operating system running in a Private Branch Exchange (PBX), features such as call forwarding (or follow me) can be added without interrupting the basic telephone service and without reconfiguring the entire telephone network.

Memory Protection

Different applications can run in different memory address spaces, which are protected from one another. If one application fails, it can corrupt only its own data but cannot corrupt the data of other applications, or of the system itself. This mechanism confines errors and prevents their propagation.

Memory Protection is elaborated further in "Protected Memory (`MEM_PROTECTED`)" on page 112.

Watchdog Timer Protection

The watchdog timer feature provides a two-tier watchdog mechanism to monitor hardware and the operating system by checking periodically that they are operating correctly. The application may also be monitored if it uses the debug-aware watchdog timer API.

Watchdog timer is elaborated further in "Watchdog Timer (`WDT`)" on page 121.

Real-Time Operation

The ChorusOS operating system provides real-time service through the following features and services, amongst others:

Synchronization	Using mutexes and real-time mutexes. See "Synchronization" on page 53 for details.
Real-time scheduling	Using pre-emptive FIFO scheduling based on thread priorities. See "First-in-First-Out Scheduling (`SCHED_FIFO`)" on page 51 for more information.
High-resolution timer	For fine-grained ordering of events and fault-detection mechanisms between nodes. See "High Resolution Timing" on page 123 for details.
Reduced context switching	
overhead and interrupt latency	By operating at the hardware register level, rather than throughout the file structure.
IPC mail boxes (MIPC)	To provide a shared message space for rapid communication between actors. See "Mailboxes (`MIPC`)" on page 64 for more information.

In addition, the ChorusOS operating system offers an implementation of the POSIX real-time API. See "POSIX Services" on page 38 for a full examination of the implementation of the POSIX real-time API.

Development Lifecycle

This section provides an overview of the stages in using the ChorusOS operating system to develop an application or system. It provides a high-level summary of the tasks described in the *ChorusOS 5.0 Application Developer's Guide*.

Installing ChorusOS

This section provides a brief overview of the installation process. For full information, see the *ChorusOS 5.0 Installation Guide*.

Installing the Development Environment on the Host

After installation is complete, the Sun Embedded Workshop software provides a development environment containing all the binary components required to build a ChorusOS operating system image. To create a system image for a particular reference target board, follow the instructions in Part II of the *ChorusOS 5.0 Installation Collection*.

Setting up a Boot Server

A boot server is a system that provides the ChorusOS operating system image for downloading to target systems. A boot server is useful if you want to make the same image available to many targets. To install an instance of the ChorusOS operating system on a boot server, follow the instructions in the *ChorusOS 5.0 Installation Guide*. The system where you installed the development environment can be used as a boot server.

Building and Booting on a Target System

When you have created an instance of the ChorusOS operating system you require, including embedded applications, and built a system image, you need to boot it on the target system. There are several ways to do this, including:

- Downloading the image at boot time from a boot server
- Loading the image from media located on the target system itself

Developing an Application

Configuring the System Image

When you develop an application, you must make sure that the instance of the ChorusOS operating system that the application will run on contains the optional components your application requires. For example, if your application uses semaphores, you must include the `SEM` option. See Appendix A for information about optional components of the ChorusOS operating system.

Writing an Application

The *ChorusOS 5.0 Application Developer's Guide* explains the following:

- General principles of developing an application that runs on the ChorusOS operating system

- Available APIs
- How to build the application
- Different ways of running the application

Tuning

When your application is written, you can create a performance profile to check for possible performance improvements. Creating a performance profile will help you to optimize the application's use of the ChorusOS operating system. See "Performance Profiling" in the *ChorusOS 5.0 Application Developer's Guide* and "Configuring and Tuning" in the *ChorusOS 5.0 System Administrator's Guide*

Developing a System

Information about advanced programming topics is not provided in this book.

- For information about porting the ChorusOS operating system software to another target, and how to add a device driver, see *ChorusOS 5.0 Board Support Package Developer's Guide*.
- For information about developing applications to use the hot restart functionality of the ChorusOS operating system, see "Recovering from Application Failure: Hot Restart" in *ChorusOS 5.0 Application Developer's Guide*
- For information about the organization of the source code and how to use it, see the *ChorusOS 5.0 Source Delivery Guide*.

Applications

Users can write their own applications in C, C++, or Java.

The ChorusOS operating system is able to load binary files of user applications from the host system acting as an NFS server, from a local disk, or from the system image itself (`/image/sys_bank`). This host-target environment allows you to load supervisor and user actors using a simple remote shell mechanism.

CHAPTER 3

ChorusOS Operating System Features

This chapter provides an introduction to all the features of the ChorusOS operating system. The features are grouped by subject area.

Basic Services

Basic and essential services are provided by the core executive API, as explained in "Core Executive" on page 32.

Core Executive API

The core executive feature API is summarized in the following table.

Function	Description
`actorCreate()`	Create an actor
`actorDelete()`	Delete an actor
`actorSelf()`	Get the current actor capability
`lapDescDup()`	Duplicate a LAP descriptor
`lapDescIsZero()`	Check a LAP descriptor
`lapDescZero()`	Clear a LAP descriptor
`lapInvoke()`	Invoke a LAP

Function	Description
`lapResolve()`	Find a LAP descriptor by name
`threadActivate()`	Activate a newly created thread
`threadContext()`	Get and/or set thread context
`threadCreate()`	Create a thread
`threadDelete()`	Delete a thread
`threadDelay()`	Delay the current thread
`threadLoadR()`	Get software register
`threadName()`	Set/Get thread symbolic name
`threadSelf()`	Get the current thread LI
`threadSemInit()`	Initialize a thread semaphore
`threadSemWait()`	Wait on a thread semaphore
`threadSemPost()`	Signal a thread semaphore
`threadStat()`	Get thread information
`threadStoreR()`	Set software register
`svExcHandler()`	Set actor exception handler
`svActorExcHandlerConnect()`	Connect actor exception handler
`svActorExctHandlerDisconnect()`	Disconnect actor exception handler
`svActorExctHandlerGetConnected()`	Get actor exception handler
`svGetInvoker()`	Get handler invoker
`svLapCreate()`	Create a LAP
`svLapDelete()`	Delete a LAP
`svMaskedLockGet()`	Disable interrupts and get a spin lock
`svMaskedLockInit()`	Initialize a spin lock
`svMaskedLockRel()`	Release a spin lock and enable interrupts
`svSpinLockGet()`	Disable preemption and get a spin lock
`svSpinLockInit()`	Initialize a spin lock
`svSpinLockRel()`	Release a spin lock and enable preemption
`svSpinLockTry()`	Try to get a spin lock and disable preemption
`svSysCtx()`	Get the system context structure address

Function	Description
`svSysPanic()`	Force panic handling processing
`svSysReboot()`	Request a reboot of the local size
`sySysTrapHandlerConnect()`	Connect a trap handler
`sySysTrapHandlerDisconnect()`	Disconnect a trap handler
`sySysTrapHandlerGetConnected()`	Get a trap handler
`Get a trap handler()`	Connect a trap handler
`svTrapDisConnect()`	Disconnect a trap handler
`sysGetConf()`	Get the ChorusOS module configuration value
`sysRead()`	Read characters from the system console
`sysReboot()`	Request a reboot of the local site
`sysWrite()`	Write characters from the system console
`sysPoll()`	Poll characters from the system console

See `CORE`(5FEA)for further details about the core executive feature.

Optional Actor Management Services

User Mode (`USER_MODE`)

The `USER_MODE` feature enables support for user mode actors that require direct access to the microkernel API.

This feature provides support for unprivileged actors, running in separate virtual user address spaces (when available).

`USER_MODE` is used in all memory models.

For details, see `USER_MODE`(5FEA).

GZ_FILE

The GZ_FILE feature enables dynamically loaded actors and dynamic libraries to be uncompressed at load time, prior to execution. This minimizes the space required to store these compressed files and the download time.

The GZ_FILE feature has no API. It is based on the gunzip tool. Thus, an executable file compressed on the host system using the gzip command (whose suffix is .gz) will be recognized automatically as a compressed executable file or dynamic library and uncompressed by the system at load time.

For details, see GZ_FILE(5FEA).

Dynamic Libraries (DYNAMIC_LIB)

The DYNAMIC_LIB feature provides support for dynamic libraries within the ChorusOS operating system. These libraries are loaded and mapped within the actor address space at execution time. Symbol resolution is performed at library load time. This feature also enables a running actor to ask for a library to be loaded and installed within its address space, and then to resolve symbols within this library. The feature handles dependencies between libraries.

The DYNAMIC_LIB feature API is summarized in the following table.

Function	**Description**
dladdr()	Translate address into symbolic information
dlclose()	Close a dynamic library
dlerror()	Get diagnostic information
dlopen()	Gain access to a dynamic library file
dlsym()	Get the address of a symbol in a dymanic library

For details, see DYNAMIC_LIB(5FEA).

Shared Libraries

Shared libraries are similar to dynamic libraries. Dynamic libraries are shared if there is no relocation in the text section. To make a dynamic library sharable, you must compile all the objects belonging to the shared library with the `FPIC = ON` imake definition. The ChorusOS operating system also provides `Imake` rules to create shared libraries.

The API which applies to dynamic libraries also applies to shared libraries. The ChorusOS operating system provides the following shared libraries for *user* actors and processes:

Library	**Description**
`libc.so`	Basic library routines
`libnsl.so`	RPC library and network resolution routine (`gethostbyname()`, and so on)
`librpc.so`	RPC library only
`libpthread.so`	POSIX thread library
`libpam.so`	Password management routines

Thread Synchronization (`MONITOR`)

Concurrent threads are synchronized by *monitors*. A monitor is a set of functions in which only one thread may execute at a time. It is possible for a thread running inside a monitor to suspend its execution so that another thread may enter the monitor. The initial thread waits for the second one to notify it (for example, that a resource is now available) and then to exit the monitor.

`MONITOR` API

The `MONITOR` API is summarized in the following table:

Function	**Description**
`monitorGet()`	Obtains the lock on the given monitor
`monitorInit()`	Initializes the given monitor
`monitorNotify()`	Notifies one thread waiting in `monitorWait()`
`monitorNotifyAll()`	Notifies all threads waiting in `monitorWait()`
`monitorRel()`	Releases a lock on a given monitor

Function	Description
`monitorWait()`	Causes a thread that owns the lock on the given monitor to suspend itself until it receives notification from another thread

POSIX Process Management API

The POSIX process management API is summarized in the following table:

Function	Description
`fork()`	Clone the current process
`pthread_atfork()`	Register `atfork()` handlers
`exec*()`	Execute a new image inside a process
`posix_spawn()`	Create a new process executing a new image
`wait()`	Wait for termination of a process
`waitpid()`	Wait for termination of a process
`_exit()`	Terminate the current process
`getpid()`	Get process identifier
`getppid()`	Get parent process identifier
`getpgid()`	Get process group identifier
`setpgid()`	Set process group identifier
`getuid()`	Get real user identifier
`geteuid()`	Get effective user identifier
`getgid()`	Get user's real group identifier
`getegid()`	Get user's effective group identifier
`getgroups()`	Get additional group identifiers
`setuid()`	Set real user identifier
`setgid()`	Set real group identifier
`seteuid()`	Set effective user identifier

Function	Description
`setegid()`	Set effective group identifier
`ptrace()`	Tracing and debugging a process

Scheduling

A scheduler is a feature that provides scheduling policies. A scheduling policy is a set of rules, procedures, or criteria used in making processor scheduling decisions. Each scheduler feature implements one or more scheduling policies, interacting with the core executive according to a defined microkernel internal interface. A scheduler is mandatory in all microkernel instances. The core executive includes the default FIFO scheduler.

All schedulers manage a certain number of per-thread and per-system parameters and attributes, and export an API for manipulation of this information or for other operations. Several system calls may be involved. For example the `threadScheduler()` system call is implemented by all schedulers, for manipulation of thread-specific scheduling attributes. Scheduling parameter descriptors defined for `threadScheduler()` are also used in the *schedparam* argument of the `threadCreate()` system call (see "Core Executive API" on page 45).

The `schedAdmin` system call is supported in some schedulers for site-wide administration of scheduling parameters.

The default scheduler present in the core executive implements a `CLASS_FIFO` scheduling class, which provides simple pre-emptive scheduling based on thread priorities.

Detailed information about these scheduling classes is found in the `threadScheduler`(2K) man page.

For details on scheduling, see the `SCHED`(5FEA) man page.

First-in-First-Out Scheduling (`SCHED_FIFO`)

The default FIFO scheduler option provides simple pre-emptive FIFO scheduling based on thread priorities. Priority of threads may vary from `K_FIFO_PRIOMAX` (0, the highest priority) to `K_FIFO_PRIOMIN` (255, the lowest priority). Within this policy, a thread becomes ready to run after being blocked is always inserted at the end of its priority-ready queue. A running thread is preempted only if a thread of a strictly higher priority is ready to run. A preempted thread is placed at the head of its priority-ready queue, so that it is selected first when the preempting thread completes or is blocked.

Multi-Class Scheduling (`SCHED_CLASS`)

The multi-class scheduler option allows multiple scheduling classes to exist simultaneously. Each active thread is subject to a single scheduling class at any one time, but can change class dynamically.

The multi-class scheduler provides the following scheduling policies:

- Priority-based round-robin, with a fixed time quantum (`CLASS_RR`).
- Real-time scheduling (`CLASS_RT`)

Round Robin Scheduling (`CLASS_RR`)

The optional `CLASS_RR` scheduling class is available only within the `SCHED_CLASS` scheduler. It is similar to `SCHED_FIFO` but implements a priority-based, preemptive policy with round-robin time slicing based on a configurable time quantum. Priority of threads may vary from `K_RR_PRIOMAX` (0, highest priority) to `K_RR_PRIOMIN` (255, lowest priority). `CLASS_RR` uses the full range of priorities (256) of the `SCHED_CLASS` scheduler.

The `SCHED_RR` policy is similar to the `SCHED_FIFO` policy, except that an elected thread is given a fixed time quantum. If the thread is still running at quantum expiration, it is placed at the end of its priority ready queue, and then may be preempted by threads of equal priority. The thread's quantum is reset when the thread becomes ready after being blocked. It is not reset when the thread is elected again after a preemption. The time quantum is the same for all priority levels and all threads. It is defined by the `K_RR_QUANTUM` value (100 milliseconds).

For details, see the `ROUND_ROBIN`(5FEA) man page.

Real-Time Scheduling (`CLASS_RT`)

The `CLASS_RT` scheduling class is available only within the `SCHED_CLASS` scheduler. It implements the same policy as the real-time class of UNIX SVR4.0. Refer to the man page of UNIX SVR4.0 for a complete description.

The real-time scheduling policy is essentially a round-robin policy, with per-thread time quanta. The priority of a thread may vary between `K_RT_PRIOMAX` (159, highest priority) and `K_RT_PRIOMIN` (100, lowest priority).

The order of priorities is inverted compared to the `CLASS_FIFO` priorities. `CLASS_RT` uses only a sub-range of the `SCHED_CLASS` priorities. This sub-range corresponds to the range [96, 155] of `CLASS_FIFO` and `CLASS_RR`.

The `CLASS_RT` manages scheduling using configurable priority default time quanta.

`SCHED` API

The `SCHED` feature API is summarized in the following table:

Function	Description	`SCHED_FIFO`	`SCHED_CLASS`
`schedAdmin()`	Administer scheduling classes		+
`threadScheduler()`	Get/set thread scheduling information	+	+

Customized Scheduling

Programmers can also develop their own scheduler to implement a specific scheduling policy.

Synchronization

The ChorusOS operating system provides the following synchronization features:

- Mutexes
- Real-time mutexes
- Semaphores
- Event flags

Mutexes (`MUTEX`)

The ChorusOS operating system provides sleep locks in the form of mutual exclusion locks, or *mutexes*. When using mutexes, threads sleep instead of spinning when contention occurs.

Mutexes are data structures allocated in the client actors' address spaces. No microkernel data structure is allocated for these objects, they are simply designated by the addresses of the structures. The number of these types of objects that threads can use is thus unlimited. Blocked threads are queued and awakened in a simple FIFO order.

`MUTEX` API

The `MUTEX` API is summarized in the following table:

Function	Description
`mutexGet()`	Acquire a mutex
`mutexInit()`	Initialize a mutex
`mutexRel()`	Release a mutex
`mutexTry()`	Try to acquire a mutex

Real-Time Mutex (`RTMUTEX`)

The real-time mutex feature, `RTMUTEX`, provides mutual exclusion locks that support priority inheritance so that priority-inversion situations are avoided. It should be noted that, although the ChorusOS operating system exports this service to applications, it does not use this mechanism for synchronizing access to the objects it manages.

For details, see the `RTMUTEX`(5FEA) man page.

`RTMUTEX` API

The `RTMUTEX` API is summarized in the following table:

Function	Description
`rtMutexGet()`	Acquire a real time mutex
`rtMutexInit()`	Initialize a real time mutex
`rtMutexRel()`	Release a real time mutex
`rtMutexTry()`	Try to acquire a real time mutex

Semaphores (SEM)

The SEM feature provides *semaphore* synchronization objects. A semaphore is an integer counter and an associated thread wait queue. When initialized, the semaphore counter receives a user-defined positive or null value.

Two main atomic operations are available on semaphores: P (or ''wait'') and V (or ''signal'').

- The counter is decremented when a thread performs a P on a semaphore. If the counter reaches a negative value, the thread is blocked and put in the semaphore's queue, otherwise, the thread continues its execution normally.
- The counter is incremented when a thread performs a V on a semaphore. If the counter is still lower than or equal to zero, one of the threads queued in the semaphore queue is picked up and awakened.

Semaphores are data structures allocated in the client actors' address spaces. No microkernel data structure is allocated for these objects, they are simply designated by the address of the structures. The number of these types of objects that threads can use is therefore unlimited.

For details, see the SEM(5FEA) man page.

SEM API

The SEM API is summarized in the following table:

Function	**Description**
`semInit()`	Initialize a semaphore
`semP()`	Wait on a semaphore
`semV()`	Signal a semaphore

Event Flags (EVENT)

The EVENT feature manages sets of event flags. An event flag set is a set of bits in memory associated with a thread-wait queue. Each bit is associated with one *event*. In this feature, the set is implemented as an unsigned integer, therefore the maximum number of flags in a set is `8*sizeof(int)`. Inside a set, each event flag is designated by an integer 0 and `8*sizeof(int)`.

When a flag is set, it is said to be *posted*, and the associated event is considered to have *occurred*. Otherwise, the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signaling purposes.

A thread can wait for a conjunctive (*and*) or disjunctive (*or*) subset of the events in one event flag set. Several threads can wait on the same event, in which case each of the threads will be made eligible to run when the event occurs.

Three functions are available on event flag sets: `eventWait()`, `eventPost()`, and `eventClear()`.

Event flag sets are data structures allocated in the client actors' address spaces. No microkernel data structure is allocated for these objects. They are simply designated by the address of the structures. The number of these types of objects that threads can use is thus unlimited.

For details, see the `EVENT`(5FEA) man page.

`EVENT` API

The `EVENT` API is summarized in the following table:

Function	**Description**
`eventClear()`	Clear event(s) in an event flag set.
`eventInit()`	Initialize an event flag set.
`eventPost()`	Signal event(s) to an event flag set.
`eventWait()`	Wait for events in an event flag set.

Communications

The ChorusOS operating system offers the following features for communications:

- Local Access Point (LAP)
- Inter-Process Communication (IPC)
- Mailboxes (MIPC)

Local Access Point (LAP)

Low overhead, same-site invocation of functions and APIs exported by supervisor actors may be executed through use of Local Access Points (LAPs). A LAP is designated and invoked via its LAP descriptor. This may be directly transmitted by a server to one or more specific client actors, via shared memory, or as an argument in another invocation.

See the `CORE`(5FEA) man page for details.

LAP Options

Optional extensions to LAP provide safe on-the-fly shutdown of local service routines and a local name binding service:

`LAPBIND`

The `LAPBIND` feature provides a nameserver from which a LAP descriptor may be requested and obtained indirectly, using a static symbolic name which may be an arbitrary character string. Using the nameserver, a LAP may be exported to any potential client that knows the symbolic name of the LAP (or of the service exported via the LAP).

The `LAPBIND` API is summarized below:

Function	Description
`lapResolve`	Find a LAP descriptor by name
`svLapBind`	Bind a name to a LAP
`svLapUnbind`	Unbind a LAP name

For details, see the `LAPBIND`(5FEA) man page.

`LAPSAFE`

The `LAPSAFE` feature does not export an API directly. It modifies the function and semantics of local access point creation and invocation. In particular, it enables the `K_LAP_SAFE` option (see `svLapCreate`(2K)), which causes validity checking to be turned on for an individual LAP. If a LAP is invalid or has been deleted, `lapInvoke()` will fail cleanly with an error return. Furthermore, the `svLapDelete()` call will block until all pending invocations have returned. This option allows a LAP to be safely withdrawn even when client actors continue to exist. It is useful for clean shutdown and reconfiguration of servers.

The `LAPSAFE` feature is a prerequisite for `HOT_RESTART`.

For details, see the `LAPSAFE`(5FEA) man page.

Inter-Process Communication (IPC)

The IPC feature provides powerful asynchronous and synchronous communication services. IPC exports the following basic communication abstractions:

- The unit of communication (message).
- Point-to-point communication endpoints (port).
- Multicast communication endpoints (groups).

Description of IPC

The IPC feature allows threads to communicate and synchronize when they do not share memory, for example, when they do not run on the same node. Communications rely on the exchange of messages through ports.

Static and Dynamic identifiers

The IPC location-transparent communication service is based on a uniform global naming scheme. Communication entities are named using global unique identifiers. Two types of global identifiers are distinguished:

- Static identifiers, provided to the system by the applications
- Dynamic identifiers, returned by the system to the application

Static identifiers are built deterministically from stamps provided by the applications. On a single site, only one communication object can be created with a given static identifier in the same communication feature. The maximum number of static stamps is fixed.

Network-wide dynamic identifiers, assigned by the system, are guaranteed to be unique across site reboots for a long time. The dynamic identifier of a new communication object is initially only known by the actor that creates the communication object. The actor can transmit this identifier to its clients through any application-specific communication mechanism (for example, in a message returned to the client).

Messages

A message is an untyped string of bytes of variable but limited size (64 KB), called the *message body*. Optionally, the sender of the message can join a second byte string to the message body, called the *message annex*. The message annex has a fixed size of 120 bytes. The message body and the message annex are transferred with `copy` semantics from the sender address space to the receiver address space.

A *current message* is associated with each thread. The current message of a thread is a system descriptor of the last message received by the thread. The current message is used when the thread has to reply to the sender of the message or acquire protection information about the sender of the message. This concept of current message allows the most common case, in which threads reply to messages in the order they are received, to be optimized and simplified. However, for other cases, the microkernel provides the facility to save the current message, and restore a previously saved message as the current message.

Ports

Messages are not addressed directly to threads, but to intermediate entities called *ports*. Ports are named using unique identifiers. A port is an address to which messages can be sent, and which has a queue holding the messages received by the port but not yet consumed by the threads. Port queues have a fixed maximum size, set as a system parameter.

For a thread to be able to consume the messages received by a port, this port must be attached to the actor that supports the thread. When a port is created by a thread, the thread attaches the port to an actor (possibly different from the one that supports the thread). The port receives a local identifier, relative to the actor to which it is attached.

A port can only be attached to a single actor at a time, but can be attached successively to different actors: a port can migrate from one actor to another. This migration can be accompanied, or not, by the messages already received by the port and not yet consumed by a thread. The concept of port provides the basis for dynamic reconfiguration. The extra level of indirection (the ports) between any two communicating threads means that the threads supplying a given service can be changed from a thread of one actor to a thread of another actor. This is done by changing the attachment of the appropriate port from the first thread's actor to the new thread's actor.

When an actor is created, a first port is attached to it automatically and is the actor's default port. The actor's default port cannot be migrated or deleted.

Groups of Ports

Ports can be assembled into groups. The concept of group extends port-to-port addressing between threads by adding a synchronous multicast facility. Alternatively, functional access to a service can be selected from among a group of (equivalent) services using port groups.

Creating a group of ports only allocates a name for the group. Ports can then be inserted into the group and it is built dynamically. A port can be removed from a group. Groups cannot contain other groups.

Like an actor, a group is named by a capability. This capability contains a unique identifier (UI), specific to the group. This UI can be used for sending messages to the ports in the group. The full group capability is needed to modify the group configuration (inserting ports in and removing ports from the group).

Like ports, messages are addressed to port groups by their UI. In the case of a group UI, the address is accompanied by an address mode. The possible address modes are:

- Broadcast to all ports in the group (`broadcast` mode).
- Addressing one of the ports of the group, selected arbitrarily (`functional` mode).
- Addressing one of the ports of the group, located on the same site as a given object designated by its UI (`associative functional` mode).
- Addressing one of the ports of the group, assuming that the selected port UI is on a different site from that of a given UI (`exclusive functional` mode).

Asynchronous and Synchronous Remote Procedure Call Communication

The IPC services allow threads to exchange messages in either `asynchronous` mode or in Remote Procedure Call (RPC) mode (demand/response mode).

`asynchronous` mode:	The sender of an asynchronous message is only blocked for the time taken for the system to process the message locally. The system does not guarantee that the message has been deposited at the destination location.
`synchronous` RPC mode:	The RPC protocol allows the construction of client-server applications, using a demand/response protocol with management of transactions. The client is blocked until a response is returned from the server, or a user-defined optional timeout occurs. RPC guarantees at-most-once semantics for the delivery of the request. It also guarantees that the response received by a client is definitely that of the server and corresponds effectively to the request (and not to a former request to which the response might have been lost). RPC also allows a client to be unblocked (with an error result) if

the server is unreachable or if the server has crashed before emitting a response. Finally, this protocol supports the propagation of abortion through the RPC. This mechanism is called *abort propagation*, that is, when a thread that is waiting for an RPC reply is aborted, this event is propagated to the thread that is currently servicing the client request.

A thread attempting to receive a message on a port is blocked until a message is received, or until a user-defined optional timeout occurs. A thread can attempt to receive a message on several ports at a time. Among the set of ports attached to an actor, a subset of enabled ports is defined. A thread can attempt to receive a message sent to any of its actor's enabled ports. Ports attached to an actor can be enabled or disabled dynamically. When a port is enabled, it receives a priority value. If several of the enabled ports hold a message when a thread attempts to receive messages on them, the port with the highest priority is selected. The actor's default port might not necessarily be enabled.

When a port is not `enabled`, it is `disabled`. This does not mean that the port cannot be used to send or receive messages. It only means that the port cannot be used in multiple-port receive requests. The default value is `disabled`.

Message Handlers

As described in the preceding section, the conventional way for an actor to consume messages delivered to its ports is for threads to express receive requests explicitly on those ports. An alternative to this scheme is the use of message handlers. Instead of creating threads explicitly, an actor can attach a handler (a routine in its address space) to the port. When a message is delivered to the port, the handler is executed in the context of a thread provided by the microkernel.

Message handlers and explicit receive requests are exclusive. When a message handler has been attached to a port, any attempt by a thread to receive a message on that port returns an error.

The use of message handlers is restricted to supervisor actors. This allows significant optimization of the RPC protocol when both the client and server reside on the same site, avoiding thread context switches (from the microkernel point of view, the client thread is used to run the handler) and memory copies (copying the message into microkernel buffers is avoided). The way messages are consumed by the threads or the handler is totally transparent to the client, the message sender. The strategy is selected by the server only.

Protection Identifiers (PI)

The IPC feature allocates a *Protection Identifier* (PI) to each actor and to each port. The structure of the Protection Identifiers is fixed, but the feature does not associate any semantics to their values. The microkernel only acts as a secure repository for these identifiers.

An actor receives, when its IPC context is initialized, a PI equal to that of the actor that created it. A port also receives a PI equal to that of the actor that created it. A system thread can change the PI of any actor or port. Subsystem process managers are in charge of managing the values given to the PI of the actors and ports they control.

When a message is sent, it is stamped with the PI of both the sending actor and its port. These values can be read by the receiver of the message, which can apply its own protection policies and thus decide whether it should reject the message. Subsystem servers can then apply the subsystem-specific protection policies, according to the PI semantics defined by the subsystem process manager.

Reconfiguration

The microkernel allows the dynamic reconfiguration of services by permitting the migration of ports. This reconfiguration mechanism requires both servers involved in the reconfiguration to be active at the same time.

The microkernel also offers mechanisms to manage the stability of the system, even in the presence of server failures. The concept of port groups is used to establish the stability of server addresses.

A port group collects several ports together. A server that possesses a port group capability can insert new ports into the group, replacing the terminated ports that were attached to servers.

A client that references a group UI (rather than directly referencing the port attached to a server) can continue to obtain the required services once a terminated port has been replaced in the group. In other words, the lifetime of a group of ports is unlimited, because groups continue to exist even when ports in the group have terminated. Logically, a group needs to contain only a single port, and this only if the server is alive. Thus clients can have stable services as long as their requests for services are made by emitting messages to a group.

Transparent IPC

Based on industry standards, transparentIPC allows applications to be distributed across multiple machines, and to run in a heterogeneous environment that comprises hardware and software with stark operational and programming incompatibilities.

At a lower level, one of the components of the ChorusOS operating system provides transparent IPC that recognizes whether a given process is available locally, or is installed on a remote system that is also running the ChorusOS operating system. When a process is accessed, IPC identifies the shortest path and quickest execution time that can be used to reach it, and communicates in a manner that makes the location entirely transparent to the application.

`IPC` *API*

The `IPC` feature API is summarized in the following table:

Function	Description
`actorPi()`	Modify the PI of an actor
`portCreate()`	Create a port
`portDeclare()`	Declare a port
`portDelete()`	Destroy a port
`portDisable()`	Disable a port
`portEnable()`	Enable a port
`portGetSeqNum()`	Get a port sequence number
`portLi()`	Acquire the local identifier (LI) of a port
`portMigrate()`	Migrate a port
`portPi()`	Modify the PI of a port
`portUi()`	Acquire the UI of a port
`grpAllocate()`	Allocate a group name
`grpPortInsert()`	Insert a port into a group
`grpPortRemove()`	Remove a port from a group
`ipcCall()`	Send synchronously
`ipcGetData()`	Get the current message body
`ipcReceive()`	Receive a message
`ipcReply()`	Reply to the current message
`ipcRestore()`	Restore a message as the current message
`ipcSave()`	Save the current message
`ipcSend()`	Send asynchronously

Function	Description
`ipcSysInfo()`	Get information about the current message
`ipcTarget()`	Construct an address
`svMsgHandler()`	Connect a message handler
`svMsgHdlReply()`	Prepare a reply to a handled message

Optional IPC Services

The ChorusOS operating system offers the following optional IPC services:

- `IPC_REMOTE`
- `Distributed IPC`

`IPC_REMOTE`

When the `IPC_REMOTE` feature is set, IPC services are provided in a distributed, location-transparent way, allowing applications distributed across the different nodes, or sites, of a network to communicate as if they were collocated on the same node.

Without this feature, IPC services can only be used in a single site.

For details, see the `IPC_REMOTE`(5FEA) man page.

Distributed IPC

The distributed IPC option extends the functionality of local IPC to provide location-transparent communication between multiple, interconnected nodes.

Mailboxes (`MIPC`)

The optional `MIPC` feature is designed to allow an application composed of one or multiple actors to create a shared communication environment (or *message space*) within which these actors can exchange messages efficiently. In particular, supervisor and user actors of the same application can exchange messages with the `MIPC` service. Furthermore, these messages can be allocated initially and sent by interrupt handlers for later processing in the context of threads.

The `MIPC` option supports the following:

- Multiple sets of dynamically allocated messages
- Messages of configurable size

Message spaces

The `MIPC` service is designed around the concept of message spaces, that encapsulates, within a single entity, both a set of message pools shared by all the actors of the application and a set of message queues through which these actors exchange messages allocated from the shared message pools.

Each message pool is defined by a pair of characteristics (*message size*, *number of messages*) provided by the application when it creates the message space. The configuration of the set of message pools depends on the communication needs of the application. From the point of view of the application, message pool requirements depend on the size range of the messages exchanged by the application, and the distribution of messages within the size range.

A message space is a temporary resource that must be created explicitly by the application. Once created, a message space can be opened by other actors of the application. A message space is bound to the actor that creates it, and it is deleted automatically when its creating actor and all actors that opened it have been deleted.

When an actor opens a message space, the system first assigns a private identifier to the message space. This identifier is returned to the calling actor and is used to designate the message space in all functions of the interface. The shared message pools are then mapped in the address space of the actor, at an address chosen automatically by the system.

Messages and queues

A message is simply an array of bytes that can be structured and manipulated at application level through any appropriate convention. Messages are presented to actors as pointers in their addressing spaces.

Messages are posted to *message queues*. Inside a message space, a queue is designated by an unsigned integer that corresponds to its index in the set of queues. Messages can also have priorities.

All resources of a message space are shared without any restriction by all actors of the application that open it. Any of these actors can allocate messages from the shared message pools. In the same way, all actors have both `send` and `receive` rights on each queue of the message space. Most applications only need to create a single message space. However, the `MIPC` service is designed to allow an application to create or open multiple message spaces. Inside these types of applications, messages cannot be exchanged directly across different message spaces. In other words, a message allocated from a message pool of one message space cannot be sent to a queue of another message space.

For details of the `MIPC` feature, see the `MIPC`(5FEA) man page.

The `MIPC` API is summarized in the following table:

Function	Description
`msgAllocate()`	Allocate a message
`msgFree()`	Free a message
`msgGet()`	Get a message
`msgPut()`	Post a message
`msgRemove()`	Remove a message from a queue
`msgSpaceCreate()`	Create a message space
`msgSpaceOpen()`	Open a message space

Drivers

The ChorusOS system provides a driver framework, allowing the third-party programmer to develop device drivers on top of a binary distribution of the ChorusOS operating system. The driver framework provides a well-defined, structured, and easy-to-use environment to develop both new drivers and client applications for existing drivers.

Host bus drivers written with the driver framework are specific to the reference target board, meaning that they are portable within that target family (UltraSPARC, PowerPC, Intel *x*86 processor families). Drivers that occupy a higher place in the hierarchical bus structure (sub-bus drivers and device drivers) are usually portable between target families.

Device Driver implementation is based on services, provided by a set of APIs, such as Peripheral Component Interconnect (PCI) or Industry Standard Architecture (ISA), which allow the developer to choose the optimizability and portability of the driver they create. This allows the driver to be written to the parent bus class, and not to the underlying platform. Drivers written within the driver framework may also take advantage of processor-specific services, allowing maximum optimization for a particular target family.

Benefits of Using the Driver Framework

Using the driver framework to build bus and device drivers in the ChorusOS operating system provides the following benefits:

- A structured framework, easing the task of building drivers

- The hierarchical structure of drivers in the driver framework mirrors the hardware structure
- Ensures compliance and functionality within the ChorusOS operating system
- Enables the user to develop multi-bus device drivers, which may run on all buses supporting the common bus driver interface
- Drivers built with the driver framework are homogeneous across various system profiles (flat memory, protected memory, virtual memory)
- Allows dynamic configuration (and reconfiguration) needed for plug-and-play, hot-plug, and hot-swap support
- Supports the binary driver model
- The APIs are version resilient
- The bus and drivers are adaptive (in terms of the memory footprint and complexity) to the various system profiles and customer requirements
- Supports the dynamic loading and unloading of driver components
- Meets real-time requirements, by providing non-blocking (asynchronous) run-time APIs

Framework Architecture Overview

In the ChorusOS operating system, a driver entity is a software abstraction of the physical bus or device. Creating a device driver using the driver framework allows the device or bus to be represented and managed by the ChorusOS operating system. The hierarchical structure of the driver software within the ChorusOS operating system mirrors the structure of the physical device or bus.

Each device or bus is represented by its own driver. A driver's component code is linked separately from the microkernel as a supervisor actor, with the device-specific code strongly localized in the corresponding device driver.

Driver components are organized, through a service-provider/user relationship, into hierarchical layers which mirror the hardware bus or device connections.

The ChorusOS operating system diver framework can be considered in two ways:

- A hierarchical set of APIs that defines the services provided for and used by each bus or device driver at each layer of the architecture. This approach ensures portability and functionality across various platforms and continued validity of drivers across subsequent system releases.
- A set of mechanisms implemented by the ChorusOS microkernel, ensuring compliance and synchronicity with the ChorusOS operating system architecture and methods.

The driver framwork architecture is shown in the following figure.

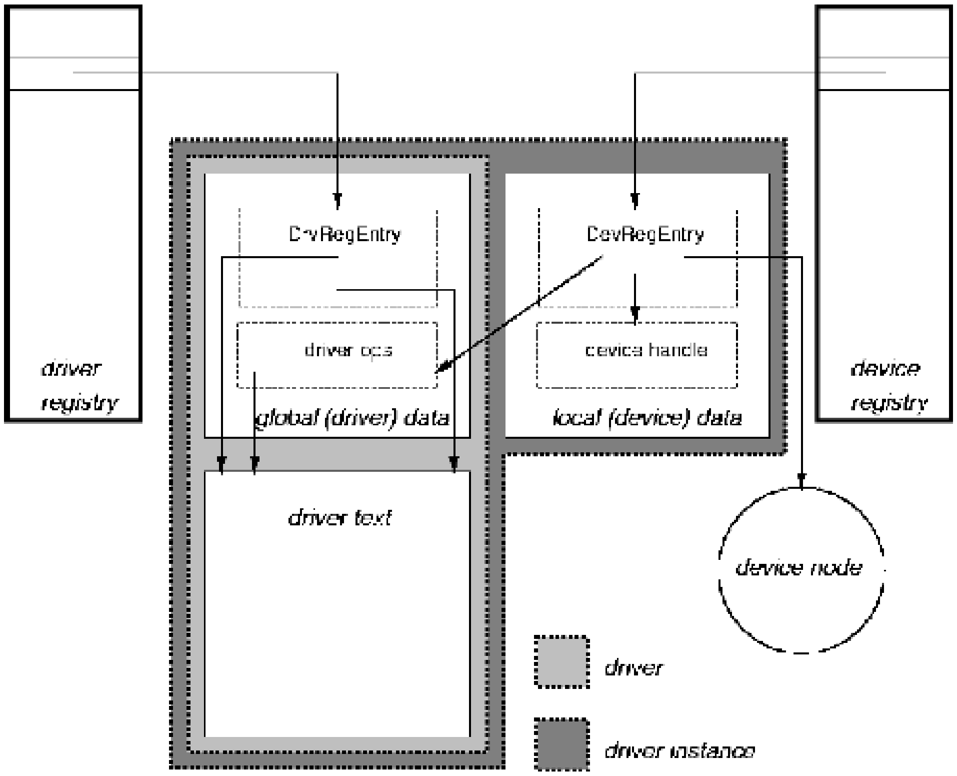

FIGURE 3–1 Driver Framerwork Architecture

For details regarding the driver framework, see the *ChorusOS 5.0 Board Support Package Developer's Guide*.

Driver Framework APIs

One of the key attributes allowing portability and modularity of devices constructed using the driver framework is the hierarchical structure of the APIs, which can also be seen as the layered interface. Within this model, all calls to the microkernel are performed through the Driver Kernel Interface (DKI) API, while all calls between drivers are handled through the Device Driver Interface (DDI) API.

Driver/Kernel Interface (DKI)

The DKI interface defines all services provided by the microkernel to driver components. Following the layered interface model, all services implemented by the DKI are called by the drivers, and take place in the microkernel.

The DKI provides two types of services:

- Common DKI services common to all platforms and processors, usable by all drivers, no matter what layer in the hierarchical model they inhabit. These services are globally designated by the `DKI` class name.
- Processor family specific DKI (FDKI) services.

Common DKI services cover:

- Synchronization through the DKI thread
- Device tree
- Driver registry
- Device registry
- General purpose memory allocation
- Timeout
- Precise busy wait
- Special-purpose physical memory allocation
- System event management
- Global interrupts masking
- Specific I/O services

Processor family specific DKI (FDKI) services cover:

- Processor interrupts management
- Processor caches management
- Processor specific I/O services
- Physical to virtual memory mapping

All DKI services are implemented as part of the embedded system library (`libebd.s.a`). Most of them are implemented as microkernel system calls. The `intro`(9DKI) man page gives an entry point to a detailed description of all DKI APIs.

DKI API

The DKI API is summarized in the following table:

Function	**Description**
`dataCacheBlockFlush()`	Cache management
`dataCacheBlockFlush_powerpc()`	PowerPC cache management

Function	Description
`dataCacheBlockInvalidate()`	Cache management
`dataCacheBlockInvalidate_powerpc()`	PowerPC cache management
`dataCacheBlockInvalidate()`	Cache management
`dataCacheBlockInvalidate_powerpc()`	PowerPC cache management
`dataCacheInvalidate()`	Cache management
`dataCacheInvalidate_powerpc()`	PowerPC cache management
`dcacheBlockFlush()`	Cache management
`dcacheBlockFlush_usparc()`	UltraSPARC cache management
`dcacheFlush()`	Cache management
`dcacheFlush_usparc()`	UltraSPARC cache management
`dcacheLineFlush()`	Cache management
`dcacheLineFlush_usparc()`	UltraSPARC cache management
`DISABLE_PREEMPT()`	Thread preemption disabling
`dtreeNodeAlloc()`	Device tree operations
`dtreeNodeChild()`	Device tree operations
`dtreeNodeDetach()`	Device tree operations
`dtreeNodeFind()`	Device tree operations
`dtreeNodeFree()`	Device tree operations
`dtreeNodeFree()`	Device tree operations
`dtreeNodePeer()`	Device tree operations
`dtreeNodeRoot()`	Device tree operations
`dtreePropAdd()`	Device tree operations
`dtreePropAlloc()`	Device tree operations
`dtreePropAttach()`	Device tree operations
`dtreePropDetach()`	Device tree operations
`dtreePropFind()`	Device tree operations
`dtreePropFindNext()`	Device tree operations
`dtreePropFree()`	Device tree operations
`dtreePropLength()`	Device tree operations

Function	Description
`dtreePropName()`	Device tree operations
`dtreePropValue()`	Device tree operations
`eieio()`	I/O services
`eieio_powerpc()`	PowerPC specific I/O services
`ENABLE_PREEMPT()`	Thread preemption enabling
`hrt()`	High Resolution Timer
`icacheBlockInval()`	Cache management
`icacheBlockInval_usparc()`	UltraSPARC cache management
`icacheInval()`	Cache management
`icacheInval_usparc()`	UltraSPARC cache management
`icacheLineInval()`	Cache management
`icacheLineInval_usparc()`	UltraSPARC cache management
`imsIntrMask_f()`	Global interrupt masking
`imsIntrUnmask_f()`	Global interrupt masking
`instCacheBlockInvalidate()`	Cache management
`instCacheBlockInvalidate_powerpc()`	PowerPC cache management
`instCacheBlockInvalidate_powerpc()`	PowerPC cache management
`instCacheInvalidate()`	Cache management
`instCacheInvalidate_powerpc()`	PowerPC cache management
`ioLoad16()`	I/O services
`ioLoad16_x86()`	Intel *x*86 specific I/O services
`ioLoad32()`	I/O services
`ioLoad32_x86()`	Intel *x*86 specific I/O services
`ioLoad8()`	I/O services
`ioLoad8_x86()`	Intel *x*86 specific I/O services
`ioRead16()`	I/O services
`ioRead16_x86()`	Intel *x*86 specific I/O services
`ioRead32()`	I/O services
`ioRead32_x86()`	Intel *x*86 specific I/O services

Function	Description
`ioRead8()`	I/O services
`ioRead8_x86()`	Intel *x*86 specific I/O services
`ioStore16()`	I/O services
`ioStore16_x86()`	Intel *x*86 specific I/O services
`ioStore32()`	I/O services
`ioStore32_x86()`	Intel *x*86 specific I/O services
`ioStore8()`	I/O services
`ioStore8_x86()`	Intel *x*86 specific I/O services
`ioWrite16()`	I/O services
`ioWrite16_x86()`	Intel *x*86 specific I/O services
`ioWrite32()`	I/O services
`ioWrite32_x86()`	Intel *x*86 specific I/O services
`ioWrite8()`	I/O services
`ioWrite8_x86()`	Intel *x*86 specific I/O services
`loadSwap_16()`	Specific I/O services
`loadSwap_32()`	Specific I/O services
`loadSwap_64()`	Specific I/O services
`loadSwapEieio_16()`	I/O services
`loadSwapEieio_16_powerpc()`	PowerPC specific I/O services
`loadSwapEieio_32()`	I/O services
`loadSwapEieio_32_powerpc()`	PowerPC specific I/O services
`loadSwap_sync_16_usparc()`	UltraSparc specific I/O services
`loadSwap_sync_32_usparc()`	UltraSparc specific I/O services
`loadSwap_sync_64_usparc()`	UltraSparc specific I/O services
`load_sync_16_usparc()`	UltraSparc specific I/O services
`load_sync_32_usparc()`	UltraSparc specific I/O services
`load_sync_64_usparc()`	UltraSparc specific I/O services
`load_sync_8_usparc()`	UltraSparc specific I/O services
`storeSwap_16()`	Specific I/O services

Function	Description
`storeSwap_32()`	Specific I/O services
`storeSwap_64()`	Specific I/O services
`storeSwapEieio_16()`	I/O services
`storeSwapEieio_16_powerpc()`	PowerPC specific I/O services
`storeSwapEieio_32()`	I/O services
`storeSwapEieio_32_powerpc()`	PowerPC specific I/O services
`storeSwap_sync_16_usparc()`	UltraSparc specific I/O services
`storeSwap_sync_32_usparc()`	UltraSparc specific I/O services
`storeSwap_sync_64_usparc()`	UltraSparc specific I/O services
`store_sync_16_usparc()`	UltraSparc specific I/O services
`store_sync_32_usparc()`	UltraSparc specific I/O services
`store_sync_64_usparc()`	UltraSparc specific I/O services
`store_sync_8_usparc()`	UltraSparc specific I/O services
`svAsyncExcepAttach()`	Asynchronous exceptions management
`svAsyncExcepAttach_usparc()`	UltraSPARC asynchronous exceptions management
`svAsyncExcepDetach_usparc()`	UltraSPARC aynchronous exceptions management
`svDeviceAlloc()`	Device registry operations
`svDeviceEntry()`	Device registry operations
`svDeviceEvent()`	Device registry operations
`svDeviceFree()`	Device registry operations
`svDeviceLookup()`	Device registry operations
`svDeviceNewCancel()`	Device registry operations
`svDeviceNewNotify()`	Device registry operations
`svDeviceRegister()`	Device registry operations
`svDeviceRelease()`	Device registry operations
`svDeviceUnregister()`	Device registry operations
`svDkiClose()`	System event management
`svDkiEvent()`	System event management

Function	Description
`svDkiInitLevel()`	Two-stage microkernel initialization
`svDkiloRemap()`	Change debug link device address
`svDkiThreadCall()`	Microkernel initialization level
`svDkiOpen()`	System event management
`svDkiThreadCall()`	Call a routine in the DKI thread context; trigger a routine in the DKI thread context; cancel a routine in the DKI thread context
`svDkiThreadCancel()`	Call a routine in the DKI thread context; trigger a routine in the DKI thread context; cancel a routine in the DKI thread context
`svDkiThreadTrigger()`	Call a routine in the DKI thread context; trigger a routine in the DKI thread context; cancel a routine in the DKI thread context
`svDriverCap()`	Driver registry operations
`svDriverEntry()`	Driver registry operations
`svDriverLookupFirst()`	Driver registry operations
`svDriverLookupNext()`	Driver registry operations
`svDriverRegister()`	Driver registry operations
`svDriverRelease()`	Driver registry operations
`svDriverUnregister()`	Driver registry operations
`svIntrAttach()`	Interrupts management
`svIntrAttach_powerpc()`	PowerPC interrupts management
`svIntrAttach_usparc()`	UltraSPARC interrupts management
`svIntrAttach_x86()`	Intel *x*86 interrupts management
`svIntrCtxGet()`	Interrupts management
`svIntrCtxGet_powerpc()`	PowerPC interrupts management
`svIntrCtxGet_usparc()`	UltraSPARC interrupts management
`svIntrCtxGet_x86()`	Intel *x*86 interrupts management
`svIntrDetach()`	Interrupts management
`svIntrDetach_powerpc()`	PowerPC interrupts management
`svIntrDetach_usparc()`	UltraSPARC interrupts management

Function	Description
`svIntrDetach_x86()`	Intel *x*86 interrupts management
`svMemAlloc()`	A general purpose memory allocator
`svMemFree()`	A general purpose memory allocator
`svPhysAlloc()`	A special purpose physical memory allocator
`svPhysFree()`	A special purpose physical memory allocator
`svPhysMap()`	Physical to virtual memory mapping
`svPhysMap_powerpc()`	PowerPC physical to virtual memory mapping
`svPhysUnmap_usparc()`	UltraSPARC physical to virtual memory mapping
`svSoftIntrAttach_usparc()`	UltraSPARC interrupts management
`svSoftIntrDetach_usparc()`	UltraSPARC interrupts management
`svTimeoutCancel()`	Timeout operations
`svTimeoutGetRes()`	Timeout operations
`svTimeoutSet()`	Timeout operations
`swap_16()`	Specific I/O services
`swap_32()`	Specific I/O services
`swap_64()`	Specific I/O services
`swapEieio_16()`	I/O services
`swapEieio_16_powerpc()`	PowerPC I/O services
`swapEieio_32()`	I/O services
`swapEieio_32_powerpc()`	PowerPC I/O services
`usecBusyWait()`	Precise busy wait service
`vmMapToPhys()`	Physical to virtual memory mapping
`vmMapToPhys_powerpc()`	PowerPC physical to virtual memory mapping
`vmMapToPhys_usparc()`	UltraSPARC physical to virtual memory mapping
`vmMapToPhys_x86()`	Intel *x*86 physical to virtual memory mapping

Device Driver Interfaces (DDI)

The DDI defines several layers of interface between different layers of device drivers in the driver hierarchy. Typically an API is defined for each class of bus or device, as a part of the DDI.

Device Driver Interface API

The DDI API is summarized in the following table:

Function	Description
`ata()`	ATA bus driver interface
`bench()`	Bench device driver interface
`bus()`	Common bus driver interface
`buscom()`	Bus communication driver interface
`busFi()`	Common bus Fault Injection driver interface
`busmux()`	Bus multiplexor driver interface
`cdrom()`	CD-ROM driver interface
`diag()`	Diagnostic driver interface
`disk()`	Hard disk device driver interface
`diskStat()`	Hard disk statistics
`ether()`	Ethernet device driver interface
`ettherStat()`	Ethernet statistics
`flash()`	Flash device driver interface
`flashCtl()`	Flash control device driver interface
`flashStat()`	Flash statistics
`generic_ata()`	Generic ATA bus master driver interface for PCI IDE devices
`gpio()`	`gpio` bus driver interface
`HSC()`	Hot swap controller driver interface
`isa()`	ISA bus driver interface
`isaFi()`	ISA fault injection driver interface
`keyboard()`	Keyboard device driver interface

Function	Description
`mngt()`	Management driver interface
`mouse()`	Mouse device driver interface
`netFrame()`	Generic representation of network frames
`pci()`	PCI bus driver interface
`pciFi()`	PCI fault injection driver interface
`pcimngr()`	PCI resource manager driver interface
`pcmcia()`	CMCIA bus driver interface
`quicc()`	QUICC bus driver interface
`ric()`	RIC device driver interface
`rtc()`	RTC device driver interface
`timer()`	TIMER device driver interface
`tx39()`	TX39 bus driver interface
`uart()`	UART device driver interface
`uartStat()`	UART statistics
`wdtimer()`	Watchdog timer device driver interface

Software Interrupts

The ChorusOS operating system DDI and DKI support software interrupts, also known as soft interrupts. Soft interrupts are not initiated by a hardware device, but rather are initiated by software. Handlers for these interrupts must also be added to and removed from the system. Soft interrupt handlers run in the interrupt context and therefore can be used to do many of the tasks that belong to an interrupt handler.

The software interrupt API (`SOFTINTR`) is summarized in the following table:

Function	Description
`svSoftIntrDeclare()`	Declares a software interrupt descriptor
`svSoftIntrTrigger()`	Triggers execution of a software interrupt
`svSoftIntrForget()`	Detaches a previously declared software interrupt

For details, see the `SOFTINTR`(5FEA) man page.

Implemented Drivers

The ChorusOS device driver framework provides many drivers. Most of these drivers, unless stated otherwise, are generic, non-platform-specific drivers and can be used regardless of platform since they use either common bus driver interface or bus-specific (not platform-specific) services.

The following drivers are implemented in the ChorusOS operating system:

Driver	Description
`amd29xxx`	`am29`*`xxx`* compatible flash driver
`atadisk`	ATA disk device driver
`benchns16550`	`ns16x50` device driver
`benchns16550`	`ns16x50` device driver
`cheerio`	Sun `cheerio` 10/100Mbps Ethernet device driver
`dec2115x`	`dec2115x` PCI-to-PCI bridges family, PCI bus driver
`dec21x4`	`dec21x4x` Ethernet device driver
`ebus`	Sun PCI/ISA bridge driver
`el3`	3Com `etherlinkIII` Ethernet device driver
`epfxxxx`	Watchdog timer device driver for devices logically programmed in Altera `epf6016/epf8020a` PLD
`epic100`	`Epic100` PCI Ethernet device driver
`falcon`	Motorola memory controller, common bus driver and flash control driver
`fccEther`	QUICC FCC controller Ethernet device driver
`generic_ata`	Generic ATA device driver for PCI based IDE controller
`i8042`	`i8042` PS/2 keyboard/mouse controller
`i8237`	Intel `i8237` DMA driver
`i8254`	Intel `i8254` timer device driver
`i8259`	Intel `i8259` timer PIC driver
`intel28F016SA`	Intel `28F016SA` compatible flash driver

Driver	Description
`intel28fxxx`	Intel `28fxxx` compatible flash driver
`isabiosisapci`	Intel `i386AT` generic ISA bus driver
`isapci`	Intel `i386AT` generic PCI/ISA bridge, ISA bus driver
`it8368e`	`IT8368E` PCMCIA controller
`m48txx`	SGS `m48txx` real time clock, NVRAM and watchdog device driver
`mc146818`	Motorola `mc146818` real time clock device driver
`ne2000`	`ne2000` Ethernet device driver
`ns16650`	Generic `ns16x50` compatible UART device driver
`pcibios`	Intel `i386AT` generic PCI bridge, PCI bus driver
`pciconf`	PCI configuration space parser driver
`pcienumo`	PCI enumerator driver
`pciFi`	PCI fault injection pseudo-driver
`pcimngr`	PCI resource manager auxiliary driver
`quicc8260`	QUICC bus driver for Motorola `mpc8260` micro-controllers
`quicc8xx`	QUICC bus driver for Motorola `mpc8xx` micro-controllers
`raven`	Motorola PCI host bridge, PCI bus driver
`ric`	Sun reset, interrupt and clock controller
`sabre`	Sun PCI host bridge, PCI bus driver
`sccEther`	QUICC SCC controller Ethernet device driver
`sccuart`	QUICC SCC controller UART device driver
`simba`	Sun advanced PCI-to-PCI bridge driver
`smc1660`	Implements the ISA Ethernet device driver interface
`smc91xx`	SMC91 family Ethernet device driver
`smcuart`	QUICC SMC controller UART device driver

Driver	Description
`tbDec`	PowerPC timebase and decrementer timer device driver
`tx3922`	TX3922 bus driver
`tx39_uart`	TX39 UART device driver
`vt82c586`	vt82c586 VIA Technologies PCI-to-ISA bridge, ISA bus driver
`vt82c586_ata`	ATA bus driver for VIA Tech VT82C586 IDE controller
`w83c553`	Winbond PCI/ISA bridge, ISA bus driver
`w83c553_ata`	ATA bus driver for Winbond W83C553 IDE controller
`z8536`	z8536/mcp750 hardware related constants
`z85x30`	Generic z85x30 hardware related constants

BSP Hot Swap

The ChorusOS Board Support Package (BSP) hot swap feature allows you to remove and replace a board from an instance of the ChorusOS operating system, without having to shut the system down. BSP hot swap starts and stops the driver corresponding to a board that is inserted or removed.

The BSP hot swap features a two-layer implementation:

- A board-dependent layer, the Hot Swap Controller (HSC), that handles the `ENUM#` signal. The `ENUM#` signal notifies the system of an insertion or a removal event specified by Peripheral Component Interconnect (PCI) hot swap.
- A common layer that implements the handler attached to the `ENUM#` signal. This is a `PciSwap` device driver installed between the bridge-specific PCI bus implementation and the implementation of the board-specific HSC hot swap interrupt driver.

Hot Swap Support

ChorusOS BSP hot swap support defines and implements a common layer between the PCI bus device driver and the HSC device driver. The implementation of the PCI bus (bridge) is chip-specific and is not supposed to be aware of the PCI hot swap capabilities. However, the handling of the `ENUM#` event is board-specific, because the `ENUM#` signal can be routed to any interrupt source and can even be polled upon timeout. The detection of this is not parent bus-specific either, but depends on the implementation of the PCI device inserted.

BSP Hot Swap support is split into three stages:

- Handling the `ENUM#` signal.
- Accessing the Hot Swap Control/Status Register (`HS_CSR`)
- Interconnecting with the System Management (system event propagation).

Compact PCI-type devices and the `dec2115`*x* bridge family are supported.

Hot Swap Sequences

The BSP hot swap feature of the ChorusOS operating system performs the following operations upon system start-up and the insertion or removal of a board.

Start up

When started, the `PciSwap` device driver looks up the device registry for the HSC device driver specified for its node. If found, the `PciSwap` driver opens the HSC device and installs its `ENUM#` handler. Without an `ENUM#` handler the PCI bus node will not support hot swap. The PCI bus driver `init-method` looks up the device registry and searches for the instance of the `PciSwap` driver specified for its device node. If found, the PCI bus driver opens the connection to this instance and installs its handlers for insertion and removal events. The PCI bus node now has hot swap capabilities.

Insertion of a Board

On insertion of a board, the `ENUM#` signal is detected and neutralized by the HSC device driver. This event is passed to the `PciSwap` driver, which detects the slot into which the board is inserted. The parent PCI bus driver is notified for each insertion. The PCI bus driver or `PciSwap` (or both) invokes the PCI enumerator. New device nodes are created or static nodes are activated. The device-specific driver establishes the connection to the PCI bus. The PCI bus driver invokes a lock method for each activated device node in the `PciSwap` driver. The slot is declared `BUSY`.

Removal of a Board

The `ENUM#` signal is detected by the HSC device driver. This event is passed to the `PciSwap` driver, which detects which slot to remove. The parent PCI bus driver is notified for each removal. The PCI bus driver sends the device shutdown event. The PCI device closes its connection to the PCI bus. The PCI bus driver or `PciSwap` (or both) invokes the PCI enumerator. The dynamic (enumerated) device nodes are deleted or static nodes are deactivated. When the last connection to the PCI bus driver for the slot is closed, the PCI bus driver invokes the method of the `PciSwap` driver. The slot is declared `FREE` and can be removed.

BSP Hot Swap API

The BSP hot swap feature API is given in the table below:

Function	Description
`open()`	Establish a connection between PCI bus and the Hot Swap Controller device.
`lock()`	Called before device initialization, to show that the PCI slot is busy.
`unlock()`	Called after device shut down, to show that the PCI slot is free to extract.
`close()`	Close a connection between PCI bus and the Hot Swap Controller device.

Hot Restart and Persistent Memory

An important benefit of the ChorusOS operating system is its hot restart capability, which provides a rapid mechanism for restarting applications or entire systems if a serious error or failure occurs.

The conventional technique, cold restart, involves rebooting or reloading an application from scratch. This causes unacceptable downtime in most systems, and there is no way to return the application to the state in which it was executing when the error occurred.

The ChorusOS hot restart feature allows execution to recommence without reloading code or data from the network or disk. When a hot-restartable process fails, persistent memory is preserved, its text and data segments are reinitialized to their original content without accessing stable storage, and the process resumes at its entry point. Hot restart is significantly faster than conventional failure recovery techniques (application reload or cold system reboot) because it protects critical information that allows the failed portions of a system to be reconstructed quickly, with minimal interruption in service. Furthermore, the hot restart technique has been applied to the entire ChorusOS operating system and not only to the applications it runs, thus ensuring a very high quality of service availability.

The ChorusOS hot restart feature addresses the high-availability requirements of ChorusOS operating system builders. Traditionally, system recovery from such errors or failures involves terminating applications and reloading them from stable storage, or rebooting the system. This causes system downtime, and can mean that important application data is lost. Such behavior is unacceptable for system builders seeking '7 by 24' or 'five nines' system availability.

The hot restart feature solves the problem of downtime and data loss by using persistent memory, that is, memory that can persist beyond the lifetime of a particular run-time instance of an actor. When an actor that uses the hot restart feature fails, or terminates abnormally, the system uses the actor data stored in persistent memory to reconstruct the actor without accessing stable storage. This reconstruction of an actor from persistent memory instead of from stable storage is known as hot restarting (or simply restarting) the actor. Persistent memory is described in detail in the following section.

Persistent Memory

The foundation of the hot restart mechanism is the use of persistent memory to store data that can persist across an actor or site restart. Persistent memory is used internally by the system to store the actor image (text and data) from which a hot restartable actor can be reconstructed. Any actor can also allocate persistent memory to store data. This data could, for example, be used to checkpoint application execution.

At the lowest level, persistent memory is a bank of memory loaded by the ChorusOS microkernel at cold boot. The content of this bank of memory is preserved across an actor or site restart. In the current implementation, the only supported medium for the persistent memory bank is RAM, that is, persistent memory is simply a reserved area of physical memory. For this reason, persistent memory resists a hot restart, but not a board reset. The size of the area of RAM reserved for persistent memory is governed by a tunable parameter.

The allocation and de-allocation (freeing) of persistent memory are managed by a ChorusOS actor known as the Persistent Memory Manager (PMM). The Persistent Memory Manager exports an API for this purpose. This API is distinct from the API used for allocating and freeing traditional memory regions. See `rgnAllocate`(2K), `rgnFree`(2K), `svPagesAllocate`(2K) and `svPagesFree`(2K) for more information on these APIs.

The Persistent Memory Manager API is described in detail in the `pmmAllocate`(2RESTART), `pmmFree`(2RESTART) and `pmmFreeAll`(2RESTART) man pages.

Hot Restart Overview

The ChorusOS hot restart feature comprises an API and run-time architecture that offer the following services:

- *Persistent memory allocation*

  The hot restart API allows actors to allocate and free portions of persistent memory while they are executing. This service is available to all ChorusOS actors after hot restart is configured.
- *Actor restart*

  With hot restart, the system is capable of detecting the abnormal termination of one or more actors and restarting them automatically from persistent memory. In addition, actors are organized into restart groups, enabling the simultaneous restart of all actors in a predefined group when a single actor in the group fails.
- *Site restart*

  With hot restart, in addition to restarting one or more actors, the system is capable of restarting all restartable actors, plus the microkernel and boot actors, for a given ChorusOS site.

The combination of these services provides a powerful framework for highly-available systems and applications, dramatically reducing the time it takes for a failed system or component to return to service.

Hot Restart API

The hot restart API is summarized in the following table:

Function	Description
`HR_EXIT_HDL()`	Macro to mark a Hot Restartable actor for clean termination
`hrfexec()`	Spawn a Hot Restartable actor
`hrfexecl()`	Spawn a Hot Restartable actor

Function	Description
`hrfexecle()`	Spawn a Hot Restartable actor
`hrfexeclp()`	Spawn a Hot Restartable actor
`hrfexecv()`	Spawn a Hot Restartable actor
`hrfexecve()`	Spawn a Hot Restartable actor
`hrfexecvp()`	Spawn a Hot Restartable actor
`hrGetActorGroup()`	Query the restart group ID for a restartable actor
`hrKillGroup()`	Kill a group of restartable actors

Restartable Actors

A restartable actor is any actor that can be restarted rapidly without accessing stable storage, when it terminates abnormally. A restartable actor is restarted from an actor image that comprises the actor's text and initialized data regions. The actor image is stored in persistent memory (unless the actor is executed in place, in which case the actor image is the actor's executable file, stored in non-persistent, physical memory). Restartable actors can use additional blocks of persistent memory to store their own data.

Figure 3–2 shows the state of a typical restartable actor at its initialization, during execution, and once it has been hot restarted as the result of an error. The actor uses persistent memory to store state data. After hot restart, the actor is reconstructed from its actor image, also in persistent memory. It is then re-executed from its initial entry point, and can retrieve the persistent state data that has been stored.

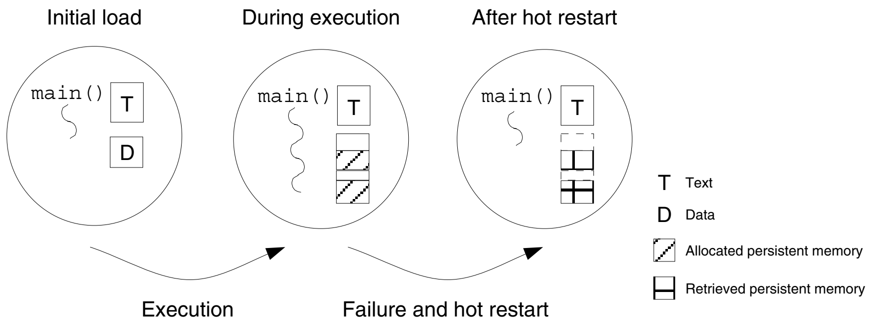

FIGURE 3–2 A typical restartable actor

In the hot restart architecture, restartable actors are managed by a ChorusOS supervisor actor called the *Hot Restart Controller* (`HR_CTRL`). The HRC monitors restartable actors to detect abnormal termination and automatically take the appropriate restart action. In the context of hot restart, abnormal termination includes unrecoverable errors, such as division by zero, a segmentation fault, unresolved page fault, or invalid operation code.

Restartable actors, like traditional ChorusOS actors, can be run in either user or supervisor mode. They can be executed from the `sysadm.ini` file, from the `C_INIT` console, or spawned dynamically during system execution. The restartable nature of restartable actors remains transparent to system actors because restartable actors do not *declare themselves* restartable, but are *run* as restartable actors. More specifically, the way in which a restartable actor is initially run determines how it will be restarted when a restart occurs:

- Restartable actors, run from the `sysadm.ini` file or directly from the `C_INIT` console, are restarted directly by the system when a restart occurs. These actors are known as *direct restartable actors*.
- Restartable actors, spawned dynamically during system execution, are restarted by the actor that initially spawned them. These actors are known as *indirect restartable actors*.

The distinction between direct and indirect restartable actors provides a useful framework for the construction of restartable groups of actors, described in "Restart Groups" on page 86.

`C_INIT` and the Hot Restart Controller provide an interface specifically for running and spawning restartable actors.

Restart Groups

Many applications are made up of not one but several actors, that cooperate to provide a service. As these actors cooperate closely, a failure in one of them can have repercussions in the others. For example, assume that actors *A* and *B* cooperate closely, using the ChorusOS operating system over IPC, and that *A* fails. Simply terminating, reloading, or hot-restarting *A* will probably not be sufficient, and will almost certainly cause *B* to fail, or to go through some special recovery action. This recovery action may in turn affect other actors that cooperate with actor *B*. Building cooperating applications that can cope with the large number of potential fault scenarios is a complex task that grows exponentially with the number of actors.

In response to this problem, the hot restart feature uses *restart groups*. A restart group is essentially a group of cooperating restartable actors that can be restarted in the event of the failure or abnormal termination of one or more actors within the group. In other words, when one actor in the group fails, all actors in the group are stopped and then restarted (either directly, by the system, or indirectly, through spawning). In this way, closely cooperating actors are guaranteed a consistent, combined operating state.

Every restartable actor in a ChorusOS operating system is a member of a restart group. Restart groups of actors are mutually exclusive and, as such, a running actor can only be a member of one actor group (declared when the actor is run), and group containment is not permitted. A restart group is created dynamically when a direct actor is declared to be a member of the group. Thus, each group contains at least one direct actor. An indirect actor is always a member of the same group as the actor that spawned it. A restart group is therefore populated through spawning from one or more direct, restartable actors.

The following figure illustrates the possible organization of restartable actors into groups within a system.

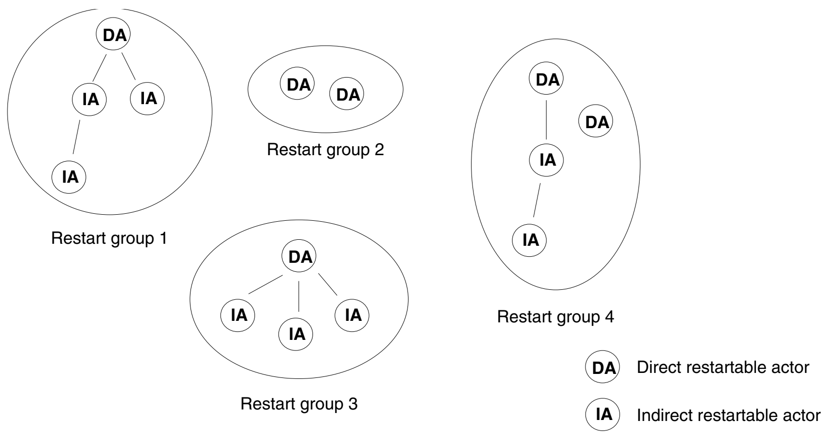

FIGURE 3–3 Restart Groups in a ChorusOS Operating System

When a group is restarted, it is restarted *from the point at which it initially started*. Figure Figure 3–4 shows the state of a group of restartable actors when the group is initially created, during execution, and when it is restarted following the failure of one of its member actors. The group contains two direct actors and one indirect (spawned) actor. The failure of the indirect actor causes a group restart. The two direct actors automatically execute their code again from their initial entry point. Time runs vertically down the page.

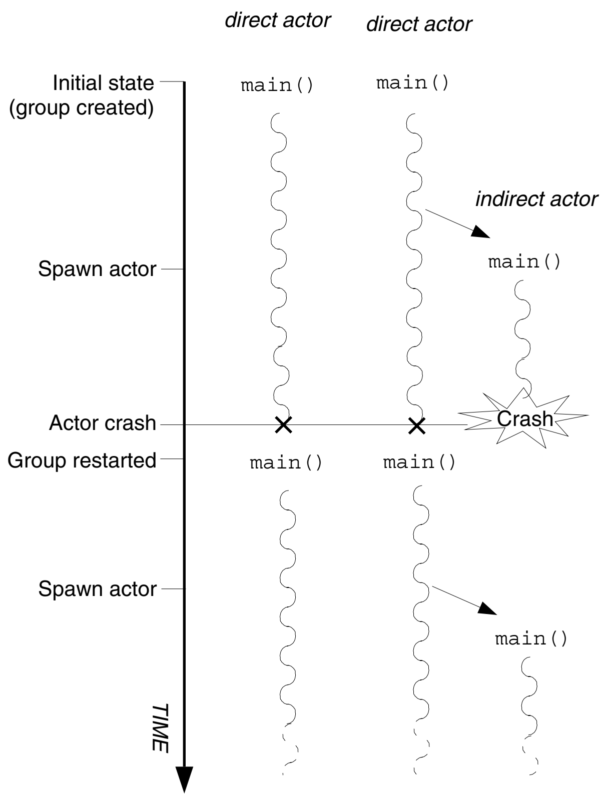

FIGURE 3–4 Group restart

Note – Simply restarting a group of actors may still not bring the system to the error-free state desired. Such a situation is possible when the failure that provokes an actor group restart is, in fact, the consequence of an error or failure elsewhere in the system. For this reason, the hot restart feature supports the concept of *site restart*, described in the next section.

Site Restart

A site restart is the reinitialization of an entire ChorusOS site (system) following the repeated failure of a group of restartable actors. It is the most severe action that can be invoked automatically by the Hot Restart Controller. A site restart involves the following:

- The microkernel and boot actors are reinitialized from the system image. This step is sometimes called a 'hot reboot' of the system, as opposed to a cold reboot, which involves a board reset and initial system loading steps. See the *ChorusOS 5.0 Board Support Package Developer's Guide* for details on a cold reboot.
- All restartable actor groups are restarted.

The precise number of group restarts to invoke a site restart is determined by the system's restart policy. The policy implemented by the hot restart feature is based on a set of system tunable parameters. You can extend the basic restart policy within your own applications, by choosing to invoke a group or site restart when particular application-specific exceptions are raised, or when particular events occur.

Hot Restart Components

The ChorusOS hot restart feature uses the following two restart-specific actors to implement hot restart services:

- A supervisor actor called the persistent memory manager (`PMM`), which offers services for allocating and freeing persistent memory blocks.
- A supervisor actor called the hot restart controller, (`HR_CTRL`). It offers the system calls that create and kill restartable actors, monitors restartable actors for abnormal termination, and takes the appropriate restart action when a failure occurs.

The Persistent Memory Manager and Hot Restart Controller principally use the services of the following:

- The `C_INIT` actor, for the interpretation of hot restart-specific commands entered on the target or host console.
- The system actor `C_OS`, solicited by the hot restart controller for loading and running restartable actors.
- The ChorusOS microkernel, for the low-level allocation of persistent memory, and for support for site restart.

The resulting architecture is summarized in Figure 3–5. Hot restart-specific components appear in gray, together with the API calls they provide. Other components appear in white. Arrows from *A* to *B* depict *A* calling functions which are implemented in *B*.

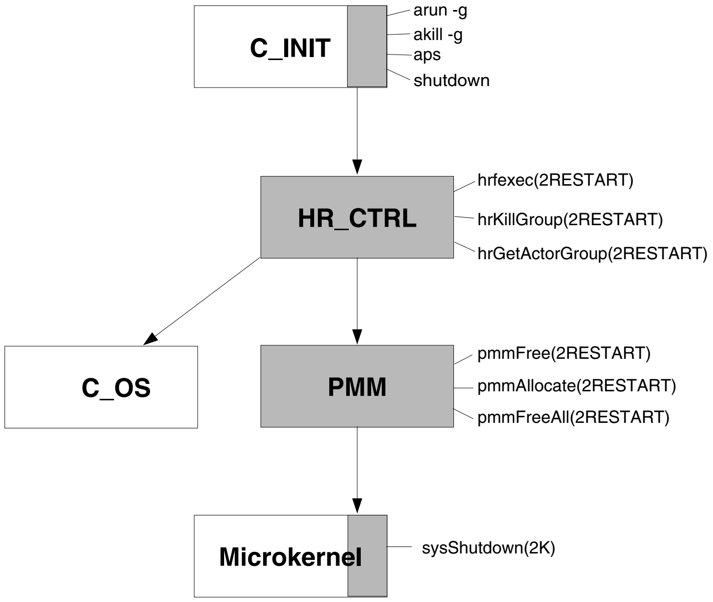

FIGURE 3–5 Hot Restart Architecture

For details of how to implement hot restart, see the *ChorusOS 5.0 Application Developer's Guide* .

POSIX Features

The ChorusOS operating system implements the following POSIX APIs:

POSIX Signals (`POSIX-SIGNALS`)

The ChorusOS operating system supports POSIX basic signal management system calls. The POSIX signals API is only available to user-mode processes.

POSIX signals API

The POSIX signals API is summarized in the following table:

Function	Description
`kill()`	Send a signal to a process
`sigemptyset()`	Set a set of signals to `NULL`
`sigfillset()`	Set all signals in a set
`sigaddset()`	Add an individual signal to a set
`sigdelset()`	Delete an individual signal from a set
`sigismember()`	Test whether a signal is member of a set
`sigaction()`	Set/Examine action for a given signal.
`pthread_sigmask()`	Set/Examine signal mask for a pthread
`sigprocmask()`	Set/Examine signal mask for a process
`sigpending()`	Examine pending signals
`sigsuspend()`	Wait for a signal
`sigwait()`	Accept a signal
`pthread_kill()`	Send a signal to a given thread
`alarm()`	Schedule delivery of an alarm signal
`pause()`	Suspend process execution
`sleep()`	Delay process execution

POSIX Real-Time Signals (`POSIX_REALTIME_SIGNALS`)

The real-time extension of POSIX signals (`POSIX_REALTIME_SIGNALS`) provides functions to send and receive queued signals. In the basic POSIX signals implementation, a particular signal is only received once by a process. Multiple occurrences of a pending signal are ignored. The real-time signals API allows multiple occurences of a signal to remain pending. POSIX real-time signals include a value that is allocated to the receiver of the signal upon reception by `sigwaitinfo()` or `sigtimedwait()`. Signals are then handled according to the value allocated to the receiver. As a consequence, the number of signals sent always corresponds to the number of signals received. This behavior is reserved for specific signals included in a special range.

POSIX Real-Time Signals API

The POSIX real-time signals API is summarized in the following table:

Function	Description
`sigqueue()`	Queue a signal to a process
`sigwaitinfo()`	Accept a signal and get info
`sigtimedwait()`	Accept a signal, wait for bounded time

POSIX Threads (`POSIX-THREADS`)

The `POSIX-THREADS` API is a compatible implementation of the POSIX 1003.1 `pthread` API.

POSIX threads API

The POSIX threads API is summarized in the following table:

Function	Description
`pthread_attr_init()`	Initialize a thread attribute object
`pthread_attr_destroy()`	Destroy a thread attribute object
`pthread_attr_setstacksize()`	Set the *stacksize* attribute
`pthread_attr_getstacksize()`	Get the *stacksize* attribute
`pthread_attr_setstackaddr()`	Set the *stackaddr* attribute
`pthread_attr_getstackaddr()`	Get the *stackaddr* attribute
`pthread_attr_setdetachstate()`	Set the *detachstate* attribute
`pthread_attr_getdetachstate()`	Get the *detachstate* attribute
`pthread_attr_setscope()`	Set the contention scope attribute
`pthread_attr_getscope()`	Get the contention scope attribute
`pthread_attr_setinheritsched()`	Set the scheduling inheritance attribute
`pthread_attr_getinheritsched()`	Get the scheduling inheritance attribute
`pthread_attr_setschedpolicy()`	Set the scheduling policy attribute

Function	Description
`pthread_attr_getschedpolicy()`	Get the scheduling policy attribute
`pthread_attr_setschedparam()`	Set the scheduling parameter attribute
`pthread_attr_getschedparam()`	Get the scheduling parameter attribute
`pthread_cancel()`	Cancel execution of a thread
`pthread_cleanup_pop()`	Pop a thread cancellation clean-up handler
`pthread_cleanup_push()`	Push a thread cancellation clean-up handler
`pthread_cond_init()`	Initialize a condition variable
`pthread_cond_destroy()`	Destroy a condition variable
`pthread_cond_signal()`	Signal a condition variable
`pthread_cond_broadcast()`	Broadcast a condition variable
`pthread_cond_wait()`	Wait on a condition variable
`pthread_cond_timedwait()`	Wait with timeout on a condition variable
`pthread_condattr_init()`	Initialize a condition variable attribute object
`pthread_condattr_destroy()`	Destroy a condition variable attribute object
`pthread_create()`	Create a thread
`pthread_equal()`	Compare thread identifiers
`pthread_exit()`	Terminate the calling thread
`pthread_join()`	Wait for thread termination
`pthread_key_create()`	Create a thread-specific data key
`pthread_key_delete()`	Delete a thread-specific data key
`pthread_kill()`	Send a signal to a thread
`pthread_mutex_init()`	Initialize a mutex
`pthread_mutex_destroy()`	Delete a mutex
`pthread_mutex_lock()`	Lock a mutex
`pthread_mutex_trylock()`	Attempt to lock a mutex without waiting
`pthread_mutex_unlock()`	Unlock a mutex
`pthread_mutexattr_init()`	Initialize a mutex attribute object
`pthread_mutexattr_destroy()`	Destroy a mutex attribute object
`pthread_once()`	Initialize a library dynamically

Function	Description
`pthread_self()`	Get the identifier of the calling thread
`pthread_setcancelstate()`	Enable or disable cancellation
`pthread_setschedparam()`	Set the current scheduling policy and parameters of a thread
`pthread_getschedparam()`	Get the current scheduling policy and parameters of a thread
`pthread_setspecific()`	Associate a thread-specific value with a key
`pthread_testcancel()`	Create cancellation point in the caller
`pthread_getspecific()`	Retrieve the thread-specific value associated with a key
`pthread_yield, sched_yield()`	Yield the processor to another thread
`sched_get_priority_max()`	Get maximum priority for policy
`sched_get_priority_min()`	Get minimum priority for policy
`sched_rr_get_interval()`	Get time quantum for `SCHED_RR` policy
`sysconf()`	Get configurable system variables

POSIX Timers (`POSIX-TIMERS`)

The `POSIX-TIMERS` API is a compatible implementation of the POSIX 1003.1 real-time clock/timer API. This feature is simply a library that might or might not be linked with an application. It is not a feature that can be turned on or off when configuring a system.

POSIX Timers API

The POSIX timers API is summarized in the following table:

Function	Description
`clock_settime()`	Set clock to a specified value
`clock_gettime()`	Get value of clock
`clock_getres()`	Get resolution of clock
`nanosleep()`	Delay the current thread with high resolution

Function	Description
`timer_create()`	Create a timer
`timer_delete()`	Delete a timer
`timer_settime()`	Set and arm or disarm a timer
`timer_gettime()`	Get remaining interval for an active timer
`timer_getoverrun()`	Get current overrun count for a timer

POSIX Message Queues (`POSIX_MQ`)

The `POSIX_MQ` feature is a compatible implementation of the POSIX 1003.1 real-time message queue API. POSIX message queues can be shared between user and supervisor processes.

POSIX Message Queue API

The POSIX message queues API is summarized in the following table:

Function	Description
`fpathconf()`	Return value of configurable limit (same as for regular files)
`mq_close()`	Close a message queue
`mq_getattr()`	Retrieve message queue attributes
`mq_open()`	Open a message queue
`mq_receive()`	Receive a message from a message queue
`mq_send()`	Send a message to a message queue
`mq_setattr()`	Set message queue attributes
`mq_unlink()`	Unlink a message queue

POSIX Semaphores (`POSIX-SEM`)

The `POSIX-SEM` API is a compatible implementation of the POSIX 1003.1 semaphores API. For general information on this feature, see the POSIX standard (IEEE Std 1003.1 - 1993). This feature is simply a library that might or might not be linked to an application. It is not a feature that can be turned on or off when configuring a system.

POSIX Semaphores API

The POSIX semaphores API is summarized in the following table. Some of the calls listed are also included in other features:

Function	Comment
`sem_open()`	Open/initialize a semaphore
`sem_close()`	Close a semaphore
`sem_init()`	Initialize a semaphore
`sem_destroy()`	Delete a semaphore
`sem_wait()`	Wait on a semaphore
`sem_trywait()`	Attempt to lock a semaphore
`sem_post()`	Signal a semaphore
`sem_getvalue()`	Get semaphore counter value
`sem_unlink()`	Remove a named semaphore

POSIX Shared Memory (`POSIX_SHM`)

The `POSIX_SHM` feature is a compatible implementation of the POSIX 1003.1 real-time shared memory objects API. For general information on this feature, see the POSIX standard (IEEE Std 1003.1b-1993).

POSIX Shared Memory API

The POSIX shared memory API is summarized in the following table. Some of the calls listed are also included in other features:

Function	Description
`close()`	Close a file descriptor
`dup()`	Duplicate an open file descriptor
`dup2()`	Duplicate an open file descriptor
`fchmod()`	Change mode of file
`fchown()`	Change owner and group of a file

Function	Description
`fcntl()`	File control
`fpathconf()`	Get configurable pathname variables
`fstat()`	Get file status
`ftruncate()`	Set size of a shared memory object
`mmap()`	Map actor addresses to memory object.
`munmap()`	Unmap previously mapped addresses
`shm_open()`	Open a shared memory object
`shm_unlink()`	Unlink a shared memory object

POSIX Sockets (`POSIX_SOCKETS`)

The `POSIX_SOCKETS` feature provides POSIX-compatible socket system calls. For general information on this feature, see the POSIX draft standard P1003.1g. The `POSIX_SOCKETS` provides support for the `AF_LOCAL`, `AF_INET`, `AF_INET6`, and `AF_ROUTE` domains. The `AF_UNIX` domain is only supported when the `AF_LOCAL` feature is present. The `AF_INET6` domain is only supported when the IPv6 feature is present.

POSIX Sockets API

The `POSIX_SOCKETS` feature API is summarized in the following table. Some of the calls listed are also included in other features:

Function	Description
`accept()`	Accept a connection on a socket
`bind()`	Bind a name to a socket
`close()`	Close a file descriptor
`connect()`	Initiate a connection on a socket
`dup()`	Duplicate an open file descriptor
`dup2()`	Duplicate an open file descriptor
`fcntl()`	File control
`getpeername()`	Get name of connected peer

Function	Description
`getsockname()`	Get socket name
`setsockopt()`	Set options on sockets
`getsockopt()`	Get options on sockets
`ioctl()`	Device control
`listen()`	Listen for connections on a socket
`read()`	Read from a socket
`recv()`	Receive a message from a socket
`recvfrom()`	Receive a message from a socket
`recvmsg()`	Receive a message from a socket
`select()`	Synchronous I/O multiplexing
`send()`	Send a message from a socket
`sendto()`	Send a message from a socket.
`sendmsg()`	Send a message from a socket
`shutdown()`	Shut down part of a full-duplex connection
`socket()`	Create an endpoint for communication
`socketpair()`	Create a pair of connected sockets
`write()`	Write on a socket

Input/Output (I/O)

When ChorusOS actors use the ChorusOS Console Input/Output API, all I/O operations (such as `printf()` and `scanf()`) will be directed to the system console of the target.

If an actor uses the ChorusOS POSIX Input/Output API and is spawned from the host with `rsh`, the standard input and output of the application will be inherited from the `rsh` program and sent to the terminal emulator on the host on which the `rsh` command was issued.

In fact, the API is the same in both cases, but the POSIX API uses a different file descriptor.

I/O Options

The ChorusOS operating system provides the following optional I/O services:

FS_MAPPER

The `FS_MAPPER` feature provides support for swap in the IOM. It requires `SCSI_DISK` to be configured, as well as `VIRTUAL_ADDRESS_SPACE` and `ON_DEMAND_PAGING`.

The `FS_MAPPER` feature exports the `swapon()` system call.

For details, see the `FS_MAPPER`(5FEA) man page.

DEV_CDROM

The `DEV_CDROM` feature provides an interface to access SCSI CD-ROM drives.

The `DEV_CDROM` feature does not itself export an API.

DEV_MEM

The `DEV_MEM` feature provides a raw interface to memory devices such as `/dev/zero`, `/dev/null`, `/dev/kmem`, and `/dev/mem`.

The `DEV_MEM` feature does not export an API itself, but allows access to the devices listed in the preceding paragraphs.

For details, see the `DEV_MEM`(5FEA) man page.

DEV_NVRAM

The `DEV_NVRAM` feature provides an interface to the `NVRAM` memory device.

For details, see the `NVRAM`(5FEA) man page.

RAM_DISK

The `RAM_DISK` feature provides an interface to chunks of memory that can be seen and handled as disks. These disks may then be initialized and used as regular file systems, although their contents will be lost at system shutdown time. This feature is also required to get access to the MS-DOS file system, which is usually embedded as part of the system boot image.

The RAM_DISK feature does not export any APIs itself.

For details, see the RAM_DISK(5FEA) man page.

FLASH

The FLASH feature provides an interface to access a memory device. The flash memory may then be formatted, labelled, and used to support regular file systems. The FLASH feature relies on the flash support based on the Flite 1.2 BSP, and is not supported for all target family architectures. See the appropriate book in the *ChorusOS 5.0 Target Platform Collection* for details of which target family architecture supports the Flite 1.2 BSP.

The FLASH feature does not itself export an API.

For details, see the FLASH(5FEA) man page.

RAWFLASH

The RAWFLASH feature provides an interface to access a raw memory device. The flash memory may then be formatted, and written to with utilities such as dd. The RAWFLASH feature is mostly used to flash the boot image onto the raw memory device.

For details, see the RAWFLASH(5FEA) man page.

VTTY

The VTTY feature provides support for serial lines on top of the BSP driver for higher levels of protocols. It is used by the PPP feature (see“Point-to-Point Protocol (PPP)” on page 138).

The VTTY feature does not itself export any APIs.

For details, see the VTTY(5FEA) man page.

SCSI_DISK

The SCSI_DISK feature provides an interface to access SCSI disks. The SCSI_DISK feature relies on the SCSI bus support provided by the BSP to access disks connected on that bus.

The `SCSI_DISK` feature does not itself export an API.

For details, see the `SCSI_DISK`(5FEA) man page.

File Systems

This section introduces the file systems supported by the ChorusOS operating system. For full details of the implementation of these file systems, see "Introduction to File System Administration for ChorusOS" in *ChorusOS 5.0 System Administrator's Guide*.

UNIX File System (`UFS`)

The UNIX file system option provides support for a disk-based file system, namely, the file system resides on physical media such as hard disks.

The UNIX file system option supports drivers for the following types of physical media:

- SCSI disks
- IDE disks
- RAM disks

The UFS feature provides POSIX-compatible file I/O system calls on top of the UFS file system on a local disk. Thus, it requires a local disk to be configured and accessible on the target system. At least one of the `RAM_DISK`, or `SCSI_DISK` features must be configured. UFS must be embedded in any configuration that exports local files through NFS.

The UFS feature API is identical to the API exported by the `NFS_CLIENT` feature. However, some system calls in this API will return with error codes since the underlying file system layout does not support all these operations. For general information on the API provided by this feature, see the POSIX standard (IEEE Std 1003.1b-1993).

UFS API

The UFS feature API is summarized in the following table. It is identical to the API exported by the `NFS_CLIENT` feature. However, some system calls in this API will return with error codes since the underlying file system layout does not support all these operations. For general information on the API provided by this feature, see the POSIX standard (IEEE Std 1003.1b-1993). Some of the calls listed are also included in other features.

Command	Description
`access`	Check access permissions
`chdir, fchdir`	Change current directory
`chflags`	Modify file flags (BSD command)
`chmod, fchmod`	Change access mode
`chown, fchown`	Change owner
`chroot`	Change root directory
`close`	Close a file descriptor
`dup, dup2`	Duplicate an open file descriptor
`fcntl`	File control
`flock`	Apply or remove an advisory lock on an open file
`fpathconf`	Get configurable pathname variables
`fsync`	Synchronize a file's in-core statistics with those on disk
`getdents`	Read directory entries
`getdirentries`	Get directory entries in a file system independent format
`getfsstat`	Get list of all mounted file systems
`ioctl`	Device control
`link`	Make a hard file link
`lseek`	Move read/write file pointer
`mkdir`	Make a directory file
`mkfifo`	Make FIFOs
`mknod`	Create a special file
`mount, umount`	Mount or unmount a file system
`open`	Open for reading or writing
`read, readv`	Read from file
`readlink`	Read a value of a symbolic link
`rename`	Change the name of a file
`revoke`	Invalidate all open file descriptors (BSD command)

Command	Description
`rmdir`	Remove a directory file
`stat, fstat, lstat`	Get file status
`statfs, fstatfs`	Get file system statistics
`symlink`	Make a symbolic link to a file
`sync`	Synchronize disk block in-core status with that on disk
`truncate, ftruncate`	Truncate a file
`umask`	Set file creation mode mask
`unlink`	Remove a directory entry
`utimes`	Set file access and modification times
`write, writev`	Write to a file

The following library calls do not support multithreaded applications:

Function	Description
`opendir()`	Open a directory
`closedir()`	Close a directory
`readdir()`	Read directory entry
`rewinddir()`	Reset directory stream
`scandir()`	Scan a directory for matching entries
`seekdir()`	Set the position of the next `readdir()` call in the directory stream
`telldir()`	Return current location in directory stream

First-in, First-Out File System (`FIFOFS`)

The `FIFOFS` feature provides support for named pipes. It requires either `NFS_CLIENT` or `UFS` to be configured as well as `POSIX_SOCKETS` and `AF_LOCAL`.

For details, see the `FIFOFS`(5FEA) man page.

FIFOFS API

The FIFOFS feature does not have its own API, but enables nodes created using mkfifo() to be used as pipes.

Network File System (NFS)

The Network File System (NFS) option provides transparent access to remote files on most UNIX (and many non-UNIX) platforms. For example, this facility can be used to load applications dynamically from the host to the target.

NFS_CLIENT

The NFS_CLIENT feature provides POSIX-compatible file I/O system calls on top of the NFS file system. It provides only the client side implementation of the protocol and thus requires a host system to provide the server side implementation of the NFS protocol. The NFS_CLIENT feature can be configured to run on top of either Ethernet or the point-to-point protocol (PPP). The NFS_CLIENT requires the POSIX_SOCKETS feature to be configured.

The NFS protocol is supported over IPv4 or IPv6 and supports both NFSv2 and NFSv3 over the user datagram protocol (UDP) and transmission control protocol (TCP).

The NFS_CLIENT feature API is summarized in the following table. For general information on the API provided by this feature, see the POSIX standard (IEEE Std 1003.1b-1993). Note that some of the calls listed are also included in other features.

Function	Description
access()	Check access permissions
chdir, fchdir()	Change current directory
chflags()	Modify file flags (BSD function)
chmod, fchmod()	Change access mode
chown, fchown()	Change owner
chroot()	Change root directory
close()	Close a file descriptor
dup, dup2()	Duplicate an open file descriptor
fcntl()	File control
flock()	Apply or remove an advisory lock on an open file

Function	Description
`fpathconf()`	Get configurable pathname variables
`fsync()`	Synchronize a file's in-core stats with those on disk
`getdents()`	Read directory entries
`getdirentries()`	Get directory entries in a file system independent format
`getfsstat()`	Get list of all mounted file systems
`ioctl()`	Device control
`link()`	Make a hard file link
`lseek()`	Move read/write file pointer
`mkdir()`	Make a directory file
`mkfifo()`	Make FIFOs
`mknod()`	Create a special file
`mount, umount()`	Mount or unmount a file system
`open()`	Open for reading or writing
`read, readv()`	Read from file
`readlink()`	Read a value of a symbolic link
`rename()`	Change the name of a file
`revoke()`	Invalidate all open file descriptors (BSD function)
`rmdir()`	Remove a directory file
`stat, fstat, lstat()`	Get file status
`statfs, fstatfs()`	Get file system statistics
`symlink()`	Make a symbolic link to a file
`sync()`	Synchronize disk block in-core status with that on disk
`truncate, ftruncate()`	Truncate a file
`umask()`	Set file creation mode mask
`unlink()`	Remove a directory entry
`utimes()`	Set file access and modification times
`write, writev()`	Write to a file

The following library calls do not support multi-threaded applications:

Function	Description
`opendir()`	Open a directory
`closedir()`	Close a directory
`readdir()`	Read directory entry
`rewinddir()`	Reset directory stream
`scandir()`	Scan a directory for matching entries
`seekdir()`	Set the position of the next `readdir()` call in the directory stream
`telldir()`	Return current location in directory stream

For details, see `NFS_CLIENT`(5FEA).

`NFS_SERVER`

The `NFS_SERVER` feature provides an NFS server on top of a local file system, most commonly UFS, but possibly `MSDOSFS`. It provides only the server side implementation of the protocol, the client side being provided by the `NFS_CLIENT` feature. The `NFS_SERVER` requires the `POSIX_SOCKETS` and `UFS` features.

The NFS protocol is supported over IPv4 and IPv6; it supports both NFSv2 and NFSv3 over UDP and TCP.

The `NFS_SERVER` feature API is summarized in the following table. For general information on the API provided by this feature, see the POSIX standard (IEEE Std 1003.1b-1993). Some of the calls listed are also included in other features.

Function	Description
`getfh()`	Get file handle
`nfssvc()`	NFS services

For details, see the `NFS_SERVER`(5FEA) man page.

MS-DOS File System (MSDOSFS)

The MSDOSFS feature provides POSIX-compatible file I/O system calls on top of the MSDOSFS file system on a local disk. This feature requires a local disk to be configured and accessible on the target system.

At least one of RAM_DISK or SCSI_DISK must be configured. It is usually embedded in any configuration which uses a file system as part of the boot image of the system. MSDOSFS is frequently used with Flash memory.

The MSDOSFS feature supports long file names and file access tables (FATs) with 12, 16, or 32-bit entries.

For details, see MSDOSFS(5FEA).

MSDOSFS *API*

The MSDOSFS feature API is identical to the API exported by the NFS_CLIENT feature. However, some system calls in this API will return with error codes since the underlying file system layout does not allow support all of these operations; for example, `symlink()` and `mknod()`. For general information on the API provided by this feature, see the POSIX standard (IEEE Std 1003.1b-1993). Some of the calls listed are also included in other features.

FS_MAPPER

The FS_MAPPER feature provides support for swap in the system. It requires either the SCSI_DISK to be configured, as well as VIRTUAL_ADDRESS_SPACE. Swap is only supported on local disks. Swapping to a remote device or file over NFS is not supported. This feature uses a dedicated file system layout on disks.

The FS_MAPPER feature exports the `swapon()` system call.

For details, see the FS_MAPPER(5FEA) man page.

ISOFS

The ISOFS file system is used to access CD_ROM media.

PROCFS

The ChorusOS operating system provides a `/proc` file system derived from the FreeBSD 4.1 implementation of `/proc`. Due to major differences in the implementation of the two systems, only a subset of the FreeBSD `/proc` file system has been retained. However, due to enhancements of the process model introduced by the ChorusOS operating system, such as the support of multi-threaded processes, extensions have been introduced to reflect the multi-threaded nature of the processes.

Such a file system is usually mounted, by convention, under the `/proc` directory. This directory is then populated and depopulated dynamically and automatically depending on the life cycle of the processes. ChorusOS actors are not reflected in this file system. Upon process creation (using `fork()` or `posix_spawn()`) an entry whose name is derived from the process identifier is created in the `/proc` directory. This per-process entry is in turn a directory whose layout is almost identical from one process to another. Threads running in this process are also represented by a regular file in the `/proc` file system (on a basis of one-to-one correspondence).

`PROCFS` *API*

The API supported by the `PROCFS` file system are similar to those exported by the UFS file system, although many of calls that do not have any significance when applied to a process will return with an error code. The list of entries that are supported below each process are listed in the following table.

Entry	**Description**
`/proc`	Mount point
`/proc/curproc`	Symbolic link to the current process
`/proc/`*xxx*	Per-process directory (where *xxx* is the PID of the process)
`/proc/`*xxx*`/file`	Symbolic link to the executable file
`/proc/`*xxx*`/stats`	Per-process instrumentation
`/proc/`*xxx*`/status`	Process status information mostly used by ps(1)
`/proc/`*xxx*`/threads/`	Process threads directory
`/proc/`*xxx*`/threads/`*tt*	Per-thread directory (where *tt* is the id of the thread)
`/proc/`*xxx*`/threads/`*tt*`/stats`	Per-thread instrumentation

`PDEVFS`

The `PDEVFS` feature is a file system that has been specifically developed for ChorusOS. By convention, it is usually mounted in the `/dev` and `/image` directories. It enables an application to create device nodes without having an actual file system such as MSDOSFS, UFS or NFS available. All data structures are maintained in memory and have to be recreated upon each reboot.

It is also used internally by the ChorusOS operating system at system startup time. File types supported are:

- Block/character devices
- Directories

Regular files and FIFOs are not supported in this PDEVFS file system.

PDEVFS *API*

The PDEVFS API is the one exported by any file system, although most of the system calls will return with an error, since only a limited subset of operations are supported and meaningful. By default, the system mounts the PDEVFS file system as the root.

Processes

The ChorusOS operating system offers the following process management services:

- Memory Management
- Time Management
- Trace Management
- Environment Variables
- Private Data
- Password Management

Memory Management

The ChorusOS operating system offers various services which enable an actor to extend its address space dynamically by allocating memory regions. An actor may also shrink its address space by freeing memory regions. The ChorusOS operating system offers the possibility of sharing an area of memory between two or more actors, regardless of whether these actors are user or supervisor actors. There are three memory management models, MEM_FLAT, MEM_PROTECTED, and MEM_VIRTUAL (see "Memory Management Models" on page 111 for details on memory management models).

Note – For some reference target boards, the ChorusOS operating system does not implement all memory management models. For example, the ChorusOS operating system for UltraSPARC IIi-based boards does not implement the `MEM_VIRTUAL` model.

Basic Concepts

Each memory management module provides semantics for subsets or variants of these concepts. These semantics and variants are introduced, but are not covered in detail, in the following sections.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor* address space and the *user* address space. A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module supports several different user address spaces, and performs memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but is only accessible to threads running with the supervisor privilege level. The microkernel code and data are located in the supervisor address space.

In addition, some privileged actors, that is, supervisor actors, also use the supervisor address space. No user address space is allocated to supervisor actors.

Regions

The address space is divided into non-overlapping regions. A region is a contiguous range of logical memory addresses, to which certain attributes are associated, such as access rights. Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created remotely in an actor other than the thread's home actor.

Protections

Regions can be created with a set of access rights or *protections*.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be set independently for sub-regions inside a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections, they are merged into one region. The programmer is warned that abusing this module could result in consuming too many of the microkernel resources associated with regions.

Memory Management Models

The model used is determined by the settings of the `VIRTUAL_ADDRESS_SPACE` and `ON_DEMAND_PAGING` features. See the `MEM`(5FEA) man page for details.

Flat memory (`MEM_FLAT`)

The microkernel and all applications run in one unique, unprotected address space. This module provides simple memory allocation services.

The flat memory module, `MEM_FLAT`, is suited for systems that do not have a memory management unit (MMU), or when use of the memory management unit is required for reasons of efficiency only.

Virtual addresses match physical addresses directly. Applications cannot allocate more memory than is physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of physically mapped memory, such as ROM, memory mapped I/O, or anywhere in RAM.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory or shared from the memory object of another region.

The concept of sharing of memory objects is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

Regions

The context of an actor is a collection of non-overlapping regions. The microkernel associates a linear buffer of physical memory to each region, consisting of a memory object. The memory object and the region have the same address and size.

It is not possible to wait for memory at the moment of creation of a region. The memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism.

Protected Memory (`MEM_PROTECTED`)

The protected memory module (`MEM_PROTECTED`) is suited to systems with memory management, address translation, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (`MEM_VIRTUAL`), it is not directly possible to use secondary storage to emulate more memory than is physically available. This module is primarily targeted at critical and non-critical real-time applications, where memory protection is mandatory, and where low-priority access to secondary storage is kept simple.

Protected memory management supports multiple address spaces and region sharing between different address spaces. However, no external segments are defined; for example, swap and on-demand paging are not supported. Access to programs or data stored on secondary devices, such as video RAM and memory-mapped I/O, must be handled by application-specific file servers.

Regions

The microkernel associates a set of physical pages with each region. This set of physical pages is called a memory object.

At the moment of creation of the region, the memory object is either allocated from free physical memory or shared with the memory object of another region. Sharing has a semantic of physical sharing.

At the moment of creation of the region, you can initialize a region from another region. This initialization has a semantic of physical allocation and copying memory at region creation time. To keep the `MEM_PROTECTED` module small, no deferred on-demand paging technique is used. An actor region maps a memory object to a given virtual address, with the associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region-creation time. The memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a software error that should be logged properly for offline analysis. It should also trigger an application-designed fault recovery procedure.

Virtual memory (`MEM_VIRTUAL`)

The virtual memory module, `MEM_VIRTUAL`, is suitable for systems with page-based memory management units and where the application programs need a high-level virtual memory management system to handle memory requirements greater than the amount of physical memory available. It supports full virtual memory, including the ability to swap memory in and out on secondary devices such as video RAM and memory-mapped I/O. The main functionalities are:

- Support of multiple, protected address spaces.
- On systems with secondary storage (the usual case), applications can use much more virtual memory than the memory physically available. This module supports full virtual memory with swapping in and out on secondary devices. This module is designed specifically to implement distributed UNIX subsystems on top of the microkernel.
- Pages are automatically swapped in and out when appropriate.

Segments

The segment is the unit of representation of information in the system.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary, with a lifetime tied to that of an actor or a thread (for example, swap objects).

The microkernel itself implements special forms of segment, such as the memory objects that are allocated along with the regions.

Like actors, segments are designated by capabilities.

Regions

An actor region maps a portion of a segment to a given virtual address with the associated access rights.

The memory management provides the mapping between regions inside an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created remotely in an actor other than the requesting actor.

Often, regions can define portions of segments that do or do not overlap. Different actors can share a segment. Segments can thus be shared across the network.

The microkernel also implements optimized region copying (copy -on-write).

Protections

Regions can be created with a set of access rights or protections.

The virtual pages constituting a memory region can be protected against certain types of access. An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be set independently for sub-regions inside a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections, they are combined into one region.

Note – Abusing the `MEM_VIRTUAL` module could result in consuming too many of the microkernel resources associated with regions.

Explicit access to a segment

Memory management also allows explicit access to segments (namely, copying) without mapping them into an address space. Object consistency is thus guaranteed during concurrent accesses on a given site. The same cache management mechanism is used for segments representing program text and data, and files accessed by conventional read/write instructions.

Optional Memory Management Features

The ChorusOS operating system offers the following optional memory management features:

`VIRTUAL_ADDRESS_SPACE`

The `VIRTUAL_ADDRESS_SPACE` feature enables separate virtual address space support using the `MEM_PROTECTED` memory management model. If this feature is disabled, all the actors and the operating system share one single, flat, address space. When this feature is enabled, a separate virtual address space is created for each user actor.

`ON_DEMAND_PAGING`

The `ON_DEMAND_PAGING` feature enables on demand memory allocation and paging using the `MEM_VIRTUAL` model. `ON_DEMAND_PAGING` is only available when the `VIRTUAL_ADDRESS_SPACE` feature is enabled.

Normally when a demand is made for memory, the same amount of physical and virtual memory is allocated by the operating system. When the `ON_DEMAND_PAGING` feature is enabled, virtual memory allocation of the user address space does not necessary mean that physical memory will be allocated. Instead, the operating system may allocate the corresponding amount of memory on a large swap disk partition. When this occurs, physical memory will be used as a cache for the swap partition.

Non-Volatile Memory (`NVRAM`)

The `NVRAM` feature provides a raw interface to non-volatile memory devices, such as `/dev/knvram` and `/dev/nvramX`.

The `NVRAM` feature does not itself export an API.

Memory Management API

The memory management API is summarized in the following table:

Function	Description	Flat	Protected	Virtual
`rgnAllocate()`	Allocate a region	+	+	+
`rgnDup()`	Duplicate an address space		+	+
`rgnFree()`	Free a region	+	+	+
`rgnInit()`	Allocate a region initialized from a segment			+
`rgnInitFromActor()`	Allocate a region initialized from another region		+	+
`rgnMap()`	Create a region and map it to a segment			+
`rgnMapFromActor()`	Allocate a region mapping another region	+	+	+
`rgnSetInherit()`	Set inheritance options for a region			+
`rgnSetPaging()`	Set paging options for a region			+
`rgnSetProtect()`	Set protection options for a region	+	+	+
`rgnStat()`	Get statistics of a region	+	+	+
`svCopyIn()`	Byte copy from user address space	+	+	+
`svCopyInString()`	String copy to user address space	+	+	+
`svCopyOut()`	Byte to user address space	+	+	+
`svPagesAllocate()`	Supervisor address space page allocator	+	+	+

Function	Description	Flat	Protected	Virtual
`svPagesFree()`	Free memory allocated by `svPagesAllocate()`	+	+	+
`svPhysAlloc()`	Physical memory page allocator	+	+	+
`svPhysFree()`	Free memory allocated by `svPhysAlloc()`	+	+	+
`svPhysMap()`	Map a physical address to the supervisor space	+	+	+
`svPhysUnMap()`	Destroy a mapping created by `svPhysMap()`	+	+	+
`svMemMap()`	Map a physical address to the supervisor space	+	+	+
`svMemUnMap()`	Destroy a mapping created by `svMemUnMap()`	+	+	+
`vmCopy()`	Copy data between address spaces	+	+	+
`vmFree()`	Free physical memory			+
`vmLock()`	Lock virtual memory in physical memory			+
`vmMapToPhys()`	Map a physical address to a virtual address		+	+
`vmPageSize()`	Get the page or block size	+	+	+
`vmPhysAddr()`	Get a physical address for a virtual address	+	+	+
`vmSetPar()`	Set the memory management parameters			+
`vmUnLock()`	Unlock virtual memory from physical memory			+

Time Management

The ChorusOS operating system provides the following time management features:

- General interval timing
- Virtual timer
- Time of day (universal time)
- Real-time Clock
- Watchdog timer
- Benchmark timing

- High resolution timing

The interrupt-level timing feature is always available and provides a traditional, one-shot, timeout service. Time-outs and the timeout granularity are based on a system-wide clock tick.

When the timer expires, a caller-provided handler is executed directly at the interrupt level. This is generally on the interrupt stack, if one exists, and with thread scheduling disabled. Therefore, the execution environment is restricted accordingly.

General Interval Timer (`TIMER`)

The `TIMER` feature implements a high-level timer service for both user and supervisor actors. One-shot and periodic timers are provided, with timeout notification via the execution of a user-provided upcall function by a handler thread in the application actor. Handler threads can invoke any microkernel or subsystem system call. This service is implemented using the `TIMEOUT` feature.

The extended timer facility uses the concept of a timer object in the actor environment. Timers are created and deleted dynamically. Once created, they are addressed by a local identifier in the context of their owning actor, and are deleted automatically when that actor terminates. Timer objects provide a naming mechanism and a locus of control for timing activities. All high-level timer operations, for example, setting, modifying, querying, or canceling pending timeouts, refer to timer objects. Timer objects are also involved in coordination with the threads used to execute application-level notification handlers.

Applications will typically use extended timer functions via a standard application-level library (see "POSIX Timers (`POSIX-TIMERS`)" on page 94). Timer handler threads are created and managed by this library. The library is expected to preallocate stack area for the notification functions, create the thread, and initialize the thread's priority, per-thread data, and all other aspects of its execution context, using standard system calls. The thread then declares itself available for execution of the tier notification (`timerThreadPoolWait`(2K)) system call to wait for the first or next relevant timeout event. Event arrival will cause the thread to return from the system call, at which point the library code can call the application's handler. The thread pool interface is designed to allow one or a small number of handler threads to service an arbitrary number of timers. An application can thus create a large number of handlers without the expense of a dedicated handler thread for each one.

At most, a single notification will be active for a given timer at any point in time. If no handler thread is available when the timer interval expires, either because the notification function is still executing from a previous expiration or because the handler thread(s) is(are) occupied executing notifications for other timers, an overrun occurs. When a handler thread becomes available (namely, re-executes `timerThreadPoolWait()`), it will return immediately and the notification function can be invoked immediately. At return from `timerThreadPoolWait()`, an overrun count is delivered to the thread. An overrun count value pertains to a particular execution of the notification function. The overrun count is defined as the number of timer expirations that have occurred since the preceding invocation of the notifying function without a handler thread being available. Thus for a periodic timer, an overrun count equal to one indicates that the current invocation was delayed, but by less than the period interval.

For details, see the `TIMER`(5FEA) man page.

`TIMER` *API*

The general interval timer (`TIMER`) API is summarized in the following table:

Function	Description
`timerThreadPoolInit()`	Initialize a thread pool
`timerThreadPoolWait()`	Wait for timer events
`timerCreate()`	Create a timer
`timerDelete()`	Delete a timer
`timerGetRes()`	Get timer resolution
`timerSet()`	Set a timer

Virtual Timer (`VTIMER`)

The `VTIMER` feature is responsible for all functions pertaining to measurement and timing of thread execution. It exports a number of functions that are used typically by higher-level operating system subsystems, such as, UNIX.

`VTIMER` functions include thread accounting (`threadTimes`(2K)) and virtual timeouts (`svVirtualTimeoutSet`(2K) and `svVirtualTimeoutCancel`(2K)). A virtual timeout handler is entered as soon as the designated thread or threads have consumed the specified amount of execution time. Virtual timeouts can be set either on individual threads, for support of subsystem-level virtual timers (for example, `SVR4`, `setitimer`, `VIRTUAL`, and `PROF` timers), or on entire actors, for support of process CPU limits.

A virtual timeout handler is entered as soon as one or more designated threads have consumed the specified amount of execution time.

Execution time accounting can be limited to execution in the thread's home actor (internal execution time), or can include cross-actor invocations such as system calls (total execution time).

The `svThreadVirtualTimeout()` and `svThreadActorTimeout()` handlers are invoked at thread level and thus can use any API service, including blocking system calls. Timeout events are delivered to application threads, such as `threadAbort()`, that is, a thread executes a virtual time handler on its own behalf.

For details about virtual time, see the `VTIMER`(5FEA) man page

`VTIMER` *API*

The virtual time API is summarized in the following table:

Function	Description
`svActorVirtualTimeoutCancel()`	Cancel an actor virtual timeout
`svActorVirtualTimeoutSet()`	Set an actor virtual timeout
`svThreadVirtualTimeoutCancel()`	Cancel a thread virtual timeout
`svThreadVirtualTimeoutSet()`	Set a thread virtual timeout
`svVirtualTimeoutCancel()`	Cancel a virtual timeout
`svVirtualTimeoutSet()`	Set a virtual timeout
`threadTimes()`	Get thread execution times
`virtualTimeGetRes()`	Get virtual time resolution

Time of Day (`DATE`)

The `DATE` feature maintains the time of day expressed in Universal Time, which is defined as the interval since 1st January 1970. Since the concept of local time is not supported directly by the operating system, time-zones and local seasonal adjustments must be handled by libraries outside the microkernel.

For details, see the `DATE`(5FEA) man page.

`DATE` *API*

The `DATE` API is summarized in the following table:

Function	Description
`univTime()`	Get time of day

Function	Description
`univTimeAdjust()`	Adjust time of day
`univTimeGetRes()`	Get time of day resolution
`univTimeSet()`	Set time of day

Date Management

ChorusOS 5.0 provides time and date management services that comply to Time Zone and Day Light Saving Time behaviors.

Date Management API

The date management utilities and API is summarized in the following table:

Function	Description
`date()`	Print or/and set the date
`settimeoftheday()`	System call to set the time of the day
`gettimeoftheday()`	System call to get the time of the day
`adjtime()`	System call to adjust the time of the day smoothly (used by the Network Time Protocol, NTP)
`ctime()`	Returns time argument as local time in ASCII string
`localtime()`	Returns time argument as local time in a structure
`gmtime()`	Returns time argument without local adjustment
`asctime()`	Returns ASCII time from time structure argument
`mktime()`	Returns time value from time structure argument
`strftime()`	Format printf like time from structure argument
`tzset()`	Set time zone information for time conversion routines

Real-Time Clock (`RTC`)

The `RTC` feature indicates whether a real-time clock (RTC) device is present on the target machine. When this feature is set and an RTC is present on the target, the `DATE` feature will retrieve time information from the RTC. If the `RTC` feature is not set, indicating an RTC is not present on the target, the `DATE` feature will emulate the RTC in software.

For details, see the `RTC`(5FEA) man page.

Watchdog Timer (`WDT`)

The watchdog timer feature enables a two-step watchdog mechanism on hardware. It consists of a lower-level system layer provided by the driver, that exposes a DDI, and a higher-level layer that hides the DDI and provides an easier API for any user program. The watchdog itself has two steps:

The interrupt step:	If the watchdog is not patted within a certain delay, an interrupt handler provided by the system is invoked. This interrupt handler attempts to shut down the system and to perform a system dump of the node to collect evidence of the problem.
The reset step:	If the interrupt step gets stuck or lasts too long, the watchdog resets the board, causing it to reboot.

The watchdog is either started by the system at system initialization or possibly by the boot loader. It is expected that a dedicated user-level process will be responsible for patting the watchdog throughout the normal life of the system. A failure in the patting process will lead to the interrupt step of the watchdog mechanism.

To cope gracefully with transitions at initialization time, as well as at system shut-down time, the system is designed to pat the watchdog by itself for a configurable amount of time at system initialization and system shut down. During these periods, where a patting process in user mode might not be possible, the system will play that role implicitly. However, the duration of these initialization and shut-down periods is bound to system configurable values, so it is impossible for initialization to reach the point where the user-level patting process begins without the watchdog interrupt occurring. Similarly, shut down is guaranteed to be bound, or the watchdog interrupt will occur.

Some hardware can support more than one watchdog. The API copes with such situations by associating handles to watchdogs. The `WDT` feature API is similar to the watchdog API for the Solaris operating environment.

For details on watchdog timer, see the `WDT`(5FEA) man page.

`WDT` *API*

The watchdog timer API is summarized in the following table:

Function	**Description**
`wdt_pat()`	Pat (reload) the watchdog timer
`wdt_alloc()`	Allocate a watchdog timer
`wdt_realloc()`	Reallocate a watchdog timer
`wdt_free()`	Disarm and free a watchdog timer
`wdt_get_maxinterval()`	Get the maximum limit (hardware) of a watchdog
`wdt_set_interval()`	Set the interval duration of a watchdog
`wdt_get_interval()`	Get the interval duration of a watchdog
`wdt_arm()`	Arm a watchdog
`wdt_disarm()`	Disarm a watchdog
`wdt_is_armed()`	Check whether a watchdog is armed
`wdt_startup_commit()`	Tells the system the initialiazation phase is over
`wdt_shutdown()`	Tells the system to start patting for shut down

Note – The `wdt_realloc()` function enables a process to regain control over a watchdog allocated by a possibly dead process.

Benchmark Timing (`PERF`)

The benchmark timing (`PERF`) feature provides a very precise measurement of short events. It is used primarily for performance benchmarking.

The `PERF` API closely follows that of the `TIMER` feature.

For details, see the `PERF`(5FEA) man page.

High Resolution Timing

The high resolution timer feature provides access to a fine-grained counter source for applications. This counter is used for functions such as fine-grained ordering of events in a node, measurements of short segments of code and fault-detection mechanisms between nodes.

The high resolution timer has a resolution better than or equal to one microsecond, and does not roll over more than once per day.

High Resolution Timer API

The high resolution timer API is summarized in the following table:

Function	**Description**
`hrTimerValue()`	Get the current value of the fine-grained timer in ticks
`hrTimerFrequency()`	Get the frequency of the fine-grained timer in Hertz
`hrTimerPeriod()`	Get the difference between the minimum and the maximum of the possible values of the fine-grained timer in ticks

Trace Management

Trace management in the ChorusOS operating system is provided by the logging, black box, system dump, and core dump features.

Logging (`LOG` and `syslog`)

The `LOG` feature provides support for logging traces into a circular buffer on a target system. This feature has always been present in the ChorusOS operating system, and is retained for backward-compatibility reasons. A new, richer service called `BLACKBOX` has been introduced and has its equivalent in the Solaris operating environment (see "Black Box (`BLACKBOX`)" on page 124).

The higher layers of the system also support a POSIX `syslog` facility. This service enables applications to write records that are marked with one of the possible predefined tags and a severity level. The records are sent to a `syslog` daemon that processes them according to a configuration file. Configuration of the daemon allows filtering of the records based on their tags and priority, and either appends them to a file, or sends them to a remote site. Records can also be ignored and discarded.

For details, see the `LOG`(5FEA) man page.

Logging API

The logging API is summarized in the following table:

Function	**Description**
`sysLog()`	Log a message in the circular buffer of the microkernel
`vsyslog()`	Write a log record (variable argument list)
`openlog()`	Open the log channel setting a default tag
`closelog()`	Close the log channel
`setlogmask()`	Set the priority mask level

In addition to the API, some other commands are provided:

Command	**Description**
`syslogd`	Daemon managing filtering and storing
`logger`	Write a message in a log
`syslogd.conf`	Configuration file for `syslogd`

Black Box (`BLACKBOX`)

The `BLACKBOX` feature provides an enhanced means for tracing and can be configured into or out of the system independently of the `LOG` feature.

The black box feature relies on multiple in-memory circular buffers that are managed by the system. One circular buffer is active at any time, which means that traces are added sequentially to that buffer. The buffer then wraps around when full. A buffer can be frozen through an explicit request indicating to the system which other buffer will be activated next. Records can be read from a frozen black box. Filtering control routines enables black box records to be discarded without the producer of such traces knowing about this filtering. Black box buffers are always part of the system dump in the case of a node failure leading to a dump.

The ChorusOS black box feature closely resembles the black box feature of the the Solaris operating environment.

For details, see `BLACKBOX`(5FEA).

Black Box API

The black box API common with the Solaris operating environment is summarized in the following table:

Function	Description
`bb_event()`	Write a record in the current black box
`bb_freeze()`	Freeze the current black box
`bb_list()`	Get the list and status of system black boxes
`bb_open()`	Open a frozen black box
`bb_read()`	Read the content of an open black box
`bb_close()`	Close an open black box
`bb_release()`	Unfreeze a frozen black box
`bb_getfilters()`	Retrieve current filters
`bb_setfilters()`	Set filters
`bb_getseverity()`	Retrieve severity level filter
`bb_setseverity()`	Set severity level filter
`bb_getprodids()`	Retrieve producer ID filter list
`bb_setprodids()`	Set producer ID filter list

The ChorusOS microkernel-specific API for `BLACKBOX` is as follows:

Function	Description
`bbEvent()`	Adds an event to the black box
`bbFreeze()`	Freezes the currently active black box and directs all future events to another black box
`bbRelease()`	Frees up a frozen black box
`bbSeverity()`	Gets and/or sets the global severity bitmap for the node

Function	Description
`bbGetNbb()`	Gets the number of black boxes configured on the node
`bbList()`	Gives information about the set of black boxes on the node
`bbFilters()`	Gets and/or sets the filter list and the filtered severity bitmap for the node
`bbProdids()`	Gets and/or sets the list of producers that have been registered to use the filter list and the filtered severity bitmap on this node
`bbOpen()`	Obtains access to a frozen black box
`bbClose()`	Releases access to a frozen black box
`bbReset()`	Resets a frozen black box
`bbName()`	Gets and/or sets the symbolic name of a persistent store used to hold the given black box

System Dump (`SYSTEM_DUMP`)

The system dump feature enables the system to collect data in case of a crash. In the ChorusOS operating system, data collection is defined as the content of the black box buffers. On system crash, these data are copied to a persistent memory area, or dump area, which is based on the `HOT_RESTART` feature of the ChorusOS operating system. The system is then hot-restarted so that the persistent memory area is preserved. This reboot operation gives control back to the ChorusOS bootMonitor, which initiates the transfer of collected data to a configurable local or remote location. Remote transfer is based on the TFTP protocol.

For details, see the `SYSTEM_DUMP`(5FEA) man page.

System Dump API

The `SYSTEM_DUMP` API is summarized in the following table:

Function	Description
`systemDumpCopy()`	Copy the black box and system information in the dump area
`systemDumpTransfer()`	Transfer the dump area to the storage location

Core Dump (`CORE_DUMP`)

The core dump feature allows offline, postmortem analysis of actors or processes that are killed by exceptions. This is performed in three steps:

- Gathering the relevant information from the actor or process to be killed
- Dumping the information into a `core` file on a stable storage medium
- Reloading the `core` file on the host machine for analysis

The `core` file is generated in the case of a fatal exception, upon request from the debugging server or agent or upon request from any actor or process. The following information is collected in the `core` file:

- The current threads and their characteristics, namely, all hardware and software registers, the scheduling characteristics, and the thread's name and ID
- The actor or process name, ID, capability, and type
- Dynamic and shared library informations, such as path names and relocation information
- Memory regions in use

For details, see the `CORE_DUMP`(5FEA) man page.

Environment Variables (`ENV`)

The ChorusOS environment variables (`ENV`) provide users and applications the ability to define configuration parameters at various stages of system construction and operation, for example at boot and run time. They also allow applications to get the values of these parameters at run time. These dynamic configuration parameters take the form of a string environment, namely, a set of string pairs (*name*, *value*).

For details, see the `ENV`(5FEA) man page.

Environment Variable API

The `ENV` API is summarized in the following table:

Function	Description
`sysGetEnv()`	Get a value.
`sysSetEnv()`	Set a value
`sysUnsetEnv()`	Delete a value

Private Data (PRIVATE-DATA)

The PRIVATE-DATA API implements a high-level interface for management of private per-thread data in the actor address space. It also provides a per-actor data service for supervisor actors only. This service is complemented by POSIX libraries, that are defined in the POSIX-THREADS(5FEA) feature, for example pthread_key_create(3POSIX) and pthread_setspecific(3POSIX).

For details, see the PRIVATE-DATA(5FEA) man page.

Private Data API

The PRIVATE-DATA API is summarized in the following table:

Function	Description
padGet()	Return actor-specific value associated with key
padKeyCreate()	Create an actor private key
padKeyDelete()	Delete an actor private key
padSet()	Set actor key-specific value
ptdErrnoAddr()	Return thread-specific *errno* address
ptdGet()	Return thread-specific value associated with key
ptdKeyCreate()	Create a thread-specific data key
ptdKeyDelete()	Delete a thread-specific data key
ptdRemoteGet()	Return a thread-specific data value for another thread
ptdRemoteSet()	Set a thread-specific data value for another thread
ptdSet()	Set a thread-specific value
ptdThreadDelete()	Delete all thread-specific values and call destructors
ptdThreadId()	Return the thread ID

Password Management

The bases for password management in the ChorusOS operating system are the classical `/etc/master.passwd` and `/etc/group` files. The ChorusOS operating system provides regular routines to access these files. These databases can be supported by either local files, NIS, or LDAP.

For details of password management, see the *ChorusOS 5.0 System Administrator's Guide*.

Password Management API

The password management API is summarized in the following table:

Function	Description
`ldap.conf()`	LDAP configuration file
`getpwuid()`	Password database operation
`getgrent()`	Group database operation
`getgrgid()`	Group database operation
`getgrnam()`	Group database operation
`setgroupent()`	Group database operation
`setgrent()`	Group database operation
`endgrent()`	Group database operation
`getpwent()`	Group database operation
`getpwnam()`	Group database operation
`getpwuid()`	Group database operation
`setpassent()`	Group database operation
`setpwent()`	Group database operation
`endpwent()`	Group database operation
`getusershell()`	Password database operation
`pwd_mkdb()`	Generate password databases
`passwd()`	Modify a user's password
`group()`	Format of the group permissions file

Administration

The administration facilities available in the ChorusOS operating system mostly consist of a set of commands activated in three different ways:

- Through a remote execution mechanism based on the remote shell feature
- Through a local command interpreter mechanism known as the `LOCAL_CONSOLE` feature
- At system start-up time, using the `sysadm.ini` file that can be embedded into the system image

Command Interpreter

In the ChorusOS operating system, commands are interpreted by the `C_INIT` actor. The `C_INIT` is loaded when the system is started and is not invoked by a user, but by the ChorusOS operating system. The `C_INIT` actor is also responsible for authentication of users that issue `C_INIT` commands.

For details, see the `C_INIT`(1M) man page.

The `C_INIT` actor offers the following options:

- Remote shell
- Local console

Remote Shell

The remote shell (`RSH`) feature gives access to `C_INIT` commands. When this feature is set, the `C_INIT` command `rshd` starts the `rsh` daemon. The `rshd` daemon is usually run from the end of the `sysadm.ini` file. It can also be run from the local console if it is available.

The `RSH` feature affects the configuration of the `C_INIT` actor. When configured, it starts running the `C_INIT` command interpreter in an `rsh` daemon thread on the target system forever. This allows a ChorusOS operating system to be administered from a host without needing to access the local console of the target system. This feature is not exclusive to the `C_INIT` `LOCAL_CONSOLE` feature. Both can be set, enabling the `C_INIT` command interpreter to be accessed either locally or remotely through the `rsh` protocol simultaneously.

See the `RSH`(5FEA) man page for details.

Remote Shell API

The `RSH` feature does not have its own API. All commands defined by `C_INIT` can be typed in on the target console. It is accessed from the host using the standard `rsh` protocol.

Local Console

This feature gives access to `C_INIT` commands through the local console of the target. When this feature is set, the `C_INIT console` command starts the command interpreter on the local console. The `console` command is usually run at the end of the `sysadm.ini` file. It can also be run through `rsh` if it is available.

See the `LOCAL_CONSOLE`(5FEA) man page for details.

Local Console API

The `LOCAL_CONSOLE` feature does not have its own API.

The `sysadm.ini` File

When the system is started, once all system components have initialized, the `C_INIT` component looks for a file named `/etc/sysadm.ini` in the embedded file system boot image. This script file is executed as the last step of the system initialization and you can customize it to run selected applications directly upon system start-up. The most usual tasks performed by `sysadm.ini` are as follows:

- Creating appropriate device entries in the `/dev` directory
- Parsing the device tree of the system and associating actual devices to correlated `/dev` entries
- Initializing the network interface, for example, to use DHCP or `ifconfig`
- Mounting local or remote file systems, or both
- Starting the local console command interpreter or the remote command interpreter, or both

System Administration Commands

The ChorusOS operating system offers a range of commands for system administration, which can be accessed via the command interpreter or included in the `sysadm.ini` file.

Command	Description
`akill`	Kills an actor
`aps`	Displays the list of all actors running on the target system
`arp`	Address resolution display and control
`arun`	Runs *actor_name* on the target system
`chat`	Automated conversational script with a modem
`chorusNSinet`	ChorusOS name servers
`chorusNSsite`	ChorusOS name servers
`chorusStat`	Print information about ChorusOS resources
`configurator`	ChorusOS configuration utility
`console`	Starts a command interpreter on the console of the target system
`cp`	Copy files
`cs`	Report the status of ChorusOS resources
`date`	Print and set the date
`dd`	Convert and copy a file
`df`	Display free disk space
`dhclient`	Dynamic Host Configuration Protocol client
`disklabel`	Read and write disk pack label
`domainname`	Set or display the name of the current YP/NIS domain
`dtree`	Displays all connected devices in the target device tree
`echo`	Echoes arguments to standard output
`env`	Displays the current environment
`ethIpcStackAttach`	Attaches the IPC stack to an Ethernet device
`flashdefrag`	Defragment a flash memory device
`format`	Format a Flash memory device
`fsck`	File system consistency check and interactive repair

Command	Description
`fsck_dos`	Create an MS-DOS (FAT) file system
`ftp`	ARPANET file transfer program
`ftpd`	Internet File Transfer Protocol server
`help`	Displays a brief message summarizing available commands
`hostname`	Set or print name of current host system
`ifconfig`	Configure network interface parameters
`ifwait`	Waits for an interface to be set up
`inetNS`	Internet name servers
`inetNSdns`	Internet name servers
`inetNShost`	Internet name servers
`inetNSien116`	Internet name servers
`inetNSnis`	Internet name servers
`ls`	List directory contents
`memstat`	Displays information about current memory usage
`mkdev`	Creates a device interface
`mkfd`	Create a bootable floppy disk from a ChorusOS boot image
`mkdir`	Create directories
`mkfifo`	Make FIFOs
`mkfs`	Replaced by `newfs`
`mkmerge`	Create a merged tree
`mknod`	Build special file
`mount`	Mount file systems
`mountd`	NFS daemon providing remote mount services
`mount_msdos`	Mount an MSDOS file system
`mount_nfs`	Mount an NFS file system
`mv`	Move files
`netstat`	Show network status

Command	Description
newfs	Construct a new file system
newfs_dos	Create an MS-DOS (FAT) file system
nfsd	NFS daemon providing remote NFS services
nfsstat	Display NFS statistics
pax	Read and write file archives and copy directory hierarchies
ping	Requests an ICMP ECHO_RESPONSE from the specified host
pppclose	Requests that pppstart daemon starts a thread to open a PPP line on device
pppd	Point-to-Point Protocol command
pppstart	Enables client PPP connections
pppstop	Disables PPP services on the target system
PROF	ChorusOS profiler server
profctl	ChorusOS profiling control tool
profrpg	ChorusOS profiling report generator
rarp	Sets the IP address of the Ethernet interface
rdbc	ChorusOS remote debugging daemon
reboot	Kills all actors on the target system
restart	Restarts the system
rm	Remove directory entries
rmdir	Remove directories
route	Manipulate the routing tables manually
rpcbind	DARPA port to RPC program number mapper
rshd	Command interpreter based on the remote shell protocol
setenv	Sets an environment variable
shutdown	Shut down and reboot or restart the system, change system state
sleep	Suspends execution of current thread
source	Reads and executes commands in file

Command	Description
`swapon`	Specify additional device for swapping
`syncd`	Update disks periodically
`sysctl`	Get or set microkernel state
`sysenv`	ChorusOS operating system environment
`telnetd`	Telnet Protocol server
`touch`	Change file access and modification times
`ulimit`	Sets or displays resource limits
`umask`	Displays or sets the file creation mask
`umount`	Unmount file systems
`uname`	Display information about the system
`unsetenv`	Sets the environment variable
`ypbind`	NIS binder process
`ypcat`	Print the values of all keys in a YP database
`ypmatch`	Print the values of one or more keys in a YP database
`ypwhich`	Return the name of the NIS server or map master

Networking

This section introduces the network protocols, libraries, and commands offered by the ChorusOS operating system. For full details of networking with the ChorusOS operating system, see the *ChorusOS 5.0 System Administrator's Guide*.

Network Protocols

The ChorusOS operating system provides TCP/IP and UDP/IP stacks (`POSIX-SOCKETS`), both over IPv4 and IPv6.

IPv4 and IPv6 can be present and used simultaneously.

IPv4

IPv4 provides the host capabilities as defined by the Internet Engineering Task Force (IETF). The following IPv4 protocols are supported:

IPv4 RFC	Description
RFC 1122	Requirements for Internet Hosts, Communication Layers
RFC 1123	Requirements for Internet Hosts, Application and Support
RFC 791	Internet Protocol
RFC 792	Internet Control Message Protocol
RFC 768	User Datagram Protocol
RFC 793	Transmission Control Protocol
RFC 2236	Internet Group Multicast Protocol
RFC 950	Internet Standard Subnetting Procedure
RFC 1058	Routing Information Protocol
RFC 1112	Host Extensions for IP Multicast
RFC 854	Telnet Protocol Specification
RFC 855	Telnet Option Specification
RFC 959	File Transfer Protocol
RFC 783	TFTP Protocol
RFC 1350	The TFTP Protocol (Revision 2)
RFC 1034	Domain Names - Concepts and Facilities
RFC 1035	Domain Names - Implementation and Specification
RFC 1055	Transmission of IP over Serial Lines
RFC 826	Address Resolution Protocol
RFC 903	A Reverse Address Resolution Protocol
RFC 1661	Point-to-Point Protocol
RFC 1570	PPP LCP Extensions
RFC 2131	Dynamic Host Configuration Protocol
RFC 951	Bootstrap Protocol

IPv4 RFC	Description
RFC 1497	BOOTP Vendor Information Extensions
RFC 1532	Clarifications and Extensions for the Bootstrap Protocol
RFC 1577	Classical IP and ARP over ATM
RFC 2453	RIP Version 2

IPv6

The following IPv6 RFCs are supported:

IPv6 RFC	Description
RFC 1981	Path MTU Discovery for IPv6
RFC 2292	Advanced Sockets API for IPv6
RFC 2373	IPv6 Addressing Architecture: supports node required addresses, and conforms to the scope requirement.
RFC 2374	An IPv6 Aggregatable Global Unicast Address Format supports 64-bit length of Interface ID
RFC 2375	IPv6 Multicast Address Assignments Userland applications use the well known addresses assigned in the RFC
RFC 2460	IPv6 specification
RFC 2461	Neighbor discovery for IPv6
RFC 2462	IPv6 Stateless Address Autoconfiguration
RFC 2463	ICMPv6 for IPv6 specification
RFC 2464	Transmission of IPv6 Packets over Ethernet Networks
RFC 2553	Basic Socket Interface Extensions for IPv6. IPv4 mapped address and special behavior of IPv6 wild card bind socket are supported
RFC 2675	IPv6 Jumbograms
RFC 2710	Multicast Listener Discovery for IPv6

The following utilities are available with IPv6 functionality:

Command	Description
`ifconfig`	Assign address to network interface and configure interface parameters
`netstat`	Symbolically displays contents of various network-related data structures
`ndp`	Symbolically displays the contents of the Neighbor Discovery cache
`route`	Manually manipulate the network routing tables
`ping6`	Elicit an `ICMP6_ECHO_REPLY` from a host or gateway
`rtsol`	Send only one Router Solicitation message to the specified interface and exit
`rtsold`	Send ICMPv6 Router Solicitation messages to the specified interfaces
`gifconfig`	Configures the physical address for the generic IP tunnel interface
`ftp`	Transfer files to and from a remote network site
`tftp`	Transfer files to and from a remote machine

For a full description of the implementation of IPv6 in the ChorusOS operating system, see "IPv6 and the ChorusOS System" in *ChorusOS 5.0 System Administrator's Guide*.

Point-to-Point Protocol (PPP)

The `PPP` feature allows serial lines to be used as network interfaces using the Point-to-Point Protocol. This feature needs to be configured for the ChorusOS operating system to fully support the various PPP-related commands provided by the ChorusOS system. These PPP-related commands are listed below:

`pppstart`:	Enables client PPP connections
`pppstop`:	Disables PPP services on the system by killing the `pppstart` daemon
`pppclose`:	Requests that the `pppstart` daemon close a previously opened PPP line
`pppd`:	Starts a PPP line

These services are complemented by `chat()`, which defines a conversational exchange between the computer and the modem. Its primary purpose is to establish the connection between the Point-to-Point Protocol daemon (`pppd`) and a remote `pppd` process.

The `PPP` feature does not export any APIs itself. It simply adds support of the PPP `ifnet` to the system.

For details, see the `PPP`(5FEA) man page.

Network Time Protocol (NTP)

The Network Time Protocol is implemented in the ChorusOS operating system as a set of daemons and commands whose purpose is to synchronize dates for different ChorusOS operating systems.

The `NTP` feature does not provide any specific API and relies on the following utilities and daemons:

`ntpd`:	Client/server daemon. The server feature provides a reference clock available to all systems on the network. The client feature is used to compute a clock according to other sources and keep the system clock synchronized with it.
`ntptrace`:	Determines where a given NTP server gets its time, and follows the chain of NTP servers back to their master time source.
`ntpq`:	The Network Time Protocol Query Program dynamically gets or sets the `ntpd` configuration.
`ntpdate`:	Sets the local date from the one provided by a remote NTP server

NTP services rely on the `adjtime()` system call.

Note – The ChorusOS operating system supports the client side of the NTP protocol (RFC 1305).

Berkley Packet Filtering (`BPF`)

The `BPF` feature provides a raw interface to data link layers in a protocol independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism. It must be configured when using the Dynamic Host Configuration Protocol (DHCP) client (`dhclient`(1M)).

For details, see the `BPF`(5FEA) man page.

DHCP

The ChorusOS operating system supports DHCP as a client and as a server. The ChorusOS boot framework has also been enhanced so that it can use the DHCP protocol to retrieve the system image and boot it on the local node, provided there is a correctly configured DHCP server on the network. The client side of DHCP is provided by the ChorusOS `dhclient`(1M) utility.

NFS

The ChorusOS operating system supports both NFSv2 and NFSv3, from client and server points of view. This is described in "Network File System (NFS)" on page 104.

NFS works over TCP or UDP on IPv4.

`IOM_IPC`

The `IOM_IPC` feature provides support for the `ethIpcStackAttach`(2K) system call and the corresponding built-in `C_INIT`(1M) command, `ethIpcStackAttach`. If the feature is not configured, the `ethIpcStackAttach`(2K) system call of the built-in `C_INIT` command will display an error message.

If the `IOM_IPC` feature is set to `true`, an IPC stack is included in the IOM system actor. The IPC stack may be attached to an Ethernet interface.

For details, see the `IOM_IPC`(5FEA) man page.

`IOM_OSI`

The `IOM_OSI` feature provides support for the `ethOSIStackAttach`(2K) system call.

If the `IOM_OSI` feature is set to `true`, an OSI stack is included in the IOM system actor. The OSI stack may be attached to an Ethernet interface.

For details, see the `IOM_OSI`(5FEA) man page.

`POSIX_SOCKETS`

The `POSIX_SOCKETS` feature is explained in "POSIX Sockets (`POSIX_SOCKETS`)" on page 97.

Network Libraries

This section describes the network libraries provided with the ChorusOS product.

RPC

The RPC library is compatible with Sun RPC, also known as ONC+. Extensions have been introduced into the library provided with the ChorusOS operating system, as well as into the Solaris operating environment, to support asynchronous communication.

The RPC library calls are available with the `POSIX_SOCKETS` feature. These calls support multithreaded applications. This feature is simply a library that might or might not be linked to an application. It is not a feature that can be turned on or off when configuring a system.

For details about RPC in the ChorusOS operating system, see `RPC`(5FEA).

LDAP

The Lightweight Directory Access Protocol (LDAP) provides access to X.500 directory services. These services can be a stand-alone part of a distributed directory service. Both synchronous and asynchronous APIs are provided. Also included are various routines to parse the results returned from these routines.

The basic interaction is as follows. Firstly, a session handle is created. The underlying session is established upon first use, which is commonly an LDAP bind operation. Next, other operations are performed by calling one of the synchronous or asynchronous search routines. Results returned from these routines are interpreted by calling the LDAP parsing routines. The LDAP association and underlying connection is then terminated. There are also APIs to interpret errors returned by LDAP server.

The LDAP API is summarized in the following table:

Function	**Description**
`ldap_add()`	Perform an LDAP adding operation
`ldap_init()`	Initialize the LDAP library
`ldap_open()`	Open a connection to an LDAP server
`ldap_get_values()`	Retrieve attribute values from an LDAP entry
`ldap_search_s()`	Perform synchronous LDAP search

Function	Description
`ldap_search_st()`	Perform synchronous LDAP search, with timeout
`ldap_abandon()`	Abandon an LDAP operation
`ldap_abandon_ext()`	Abandon an LDAP operation
`ldap_delete_ext()`	Perform an LDAP delete operation
`ldap_delete_ext_s()`	Perform an LDAP delete operation synchronously
`ldap_control_free()`	Dispose of a single control or an array of controls allocated by other LDAP APIs
`ldap_controls_free()`	Dispose of a single control or an array of controls allocated by other LDAP APIs
`ldap_extended_operation_s()`	
`ldap_msgtype()`	Returns the type of an LDAP message
`ldap_msgid()`	Returns the ID of an LDAP message
`ldap_count_values()`	Count number of values in an array
`ldap_explode_dn()`	Takes a domain name (DN) as returned by `ldap_get_dn()` and breaks it into its component parts
`ldap_dn2ufn()`	Turn a DN as returned by `ldap_get_dn()` into a more user- friendly form
`ldap_explode_dns()`	Take a DNS-style DN and break it up into its component parts
`ldap_dns_to_dn()`	Converts a DNS domain name into an X.500 distinguished name
`ldap_value_free()`	Free an array of values
`ldap_is_dns_dn()`	Returns non-zero if the DN string is an experimental DNS-style DN
`ldap_explode_rdn()`	Breaks an RDN into its component parts
`ldap_bind()`	Perform an LDAP bind operation
`ldap_bind_s()`	Perform an LDAP bind operation synchronously
`ldap_simple_bind()`	Initiate asynchronous bind operation and return message ID of the request sent

Function	Description
`ldap_simple_bind_s()`	Initiate synchronous bind operation and return message ID of the request sent
`ldap_sasl_cram_md5_bind_s()`	General and extensible authentication over LDAP through the use of the Simple Authentication Security Layer (SASL)
`ldap_init()`	Allocates an LDAP structure but does not open an initial connection
`ldap_modify_ext_s()`	Perform an LDAP modify operation
`ldap_modrdn_s()`	Perform an LDAP modify RDN operation synchronously
`ldap_search()`	Perform LDAP search operations

For details, see the `ldap`(3LDAP) man page.

FTP

The FTP utility is the user interface to the ARPANET standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The FTP API is summarized in the following table:

Function	Description
`ftpd()`	Internet File Transfer Protocol server
`ftpdStartSrv()`	Initializes FTP service
`ftpdHandleCnx()`	Manages an FTP connection
`lreply()`	Reply to an FTP client
`perror_reply()`	Reply to an FTP client
`reply()`	Reply to an FTP client
`ftpdGetCnx()`	Accepts a new FTP connection
`ftpdOob()`	Check for out-of-band data on the control connection

For details, see the *ChorusOS man pages section 3FTPD: FTP Daemon Library*.

Telnet

You can perform remote login operations on the ChorusOS operating system using the Telnet virtual terminal protocol. The Telnet API is summarized in the following table:

Function	Description
`inetAccept()`	Wait for a new INET connection
`inetBind()`	Bind, close INET sockets
`inetClient()`	Wait for a new INET connection
`inetClose()`	Bind, close INET socket
`telnetdFlush()`	Write or flush a Telnet session
`telnetdFree()`	Initialize or free a Telnet session
`telnetdGetTermState()`	Get or set Telnet terminal state
`telnetdInit()`	Initialize or free a Telnet session
`telnetdPasswd()`	Telnet session authentication
`telnetdRead()`	Read from a Telnet session
`telnetdReadLine()`	Read a line of characters from a Telnet session
`telnetdSetTermState()`	Get or set Telnet terminal state
`telnetdUser()`	Telnet session authentication
`telnetdWrite()`	Write or flush a Telnet session

See the *ChorusOS man pages section 3TELD: Telnet Services*.

Network Commands

The ChorusOS operating system offers the following network commands:

TABLE 3–1 ChorusOS Network Commands

Command	IPv4 Compatible	IPv6 Compatible
`arp`	Yes	N/A
`ftp`	Yes	Yes
`ftpd`	Yes	No

TABLE 3–1 ChorusOS Network Commands *(Continued)*

Command	IPv4 Compatible	IPv6 Compatible
`gifconfig`	Yes	Yes
`ifconfig`	Yes	Yes
`ndp`	No	Yes
`netstat`	Yes	Yes
`nfsd`	Yes	No
`nfsstat`	Yes	No
`ping`	Yes	N/A
ping6	N/A	Yes
`pppstart`	Yes	No
`route`	Yes	Yes
`rpcbind`	Yes	Yes
rpcinfo	Yes	Yes
`teld`	Yes	No
`tftpd`	Yes	Yes
`ypcat`	Yes	No
`ypmatch`	Yes	No
`ypwhich`	Yes	No
`dhclient`	Yes	No
`dhcpd`	Yes	No
`ntpd`	Yes	No
`ntpdate`	Yes	No
`ntpq`	Yes	No
`tcpdump`	Yes	Yes
`rtsol`	N/A	Yes
`rtsold`	N/A	Yes
`traceroute`	Yes	No

Naming Services

Naming services in the ChorusOS operating system are provided by DNS and NIS.

The Domain Name System (DNS) commands provide a standard, stable and robust architecture used for the naming architecture on the Internet Protocol. DNS is used widely on the Internet.

Name resolution is ensured by DNS servers (named), one of which is the primary server. This server reads the name records stored in a database on disk (this database file is managed by the administrator). The other servers are secondary, which means that they acquire the name records from the primary server, and do not read them from the main database file. However, these secondary servers may store records in a cache file on disk to improve restart performances. These cache files are not intended to be edited manually. The user program performs the name resolution by sending queries to DNS name servers. Generally, each host is configured such that it knows the addresses of all name servers (primary and secondary).

The ChorusOS operating system can also be bound to a Network Information Service (NIS) database.

The naming service API is summarized in the following table:

Command	Description
`named`	DNS server
`named-xfer`	Perform an inbound zone transfer
`gethostbyname`	Convert name into IP address
`gethostbyaddress`	Convert IP address into name
`gethostbyname2`	Perform lookups in address families other than `AF_INET`
`gethostbyaddr`	Get network host entry from IP address
`gethostent`	Reads `/etc/hosts`, and opens file if necessary
`sethostent`	Opens and/or rewinds `/etc/hosts`
`endhostent`	Closes the file
`herror`	Print an error message describing a failure
`hstrerror`	Returns a string which is the message text corresponding to the value of the err parameter

Command	Description
`getaddrinfo`	Protocol-independent nodename-to-address translation
`freeaddrinfo`	Frees structure pointed to by the *ai* argument
`gai_strerror`	Returns a pointer to a string describing a given error code
`getnetent`	Get network entry
`getnetbyaddr`	Search for net name by address
`getnetbyname`	Search for net address by name
`setnetent`	Opens and rewinds the file
`endnetent`	Closes the file

System Instrumentation

The ChorusOS operating system provides instrumentation to inform applications of the current use of the various resources managed by the system. Several kinds of instrumentation are exported by the system:

Attributes:	Static read-only values that show how the system is configured. These attributes are usually tunable values set when you build your system.
Counters:	Values that increase constantly, such as, the number of bytes transferred on a disk, or the number of packets received on a network interface. Such counters can only be read by the application. Some counters can be reset.
Gauges:	Values that increase and decrease depending upon the activity of the system, such as, the amount of memory used or the number of open file descriptors used. Most of the time, gauges are associated with watermarks. The ChorusOS operating system manages one high and one low watermark per gauge. Gauges can only be read, while watermarks can be read or reset.
Thresholds:	Gauges with watermarks can also be associated with either a high or a low threshold, depending upon the semantics of the resource being instrumented. A threshold is represented by two values: ■ a *rise* value, such that when the gauge's value passes the rise value a system event will be generated and posted to the

application level
- a *clear* value, such that when the gauge's value passes the clear value, another system event will be generated and posted to application level

Rise and clear values are illustrated in the following figures:

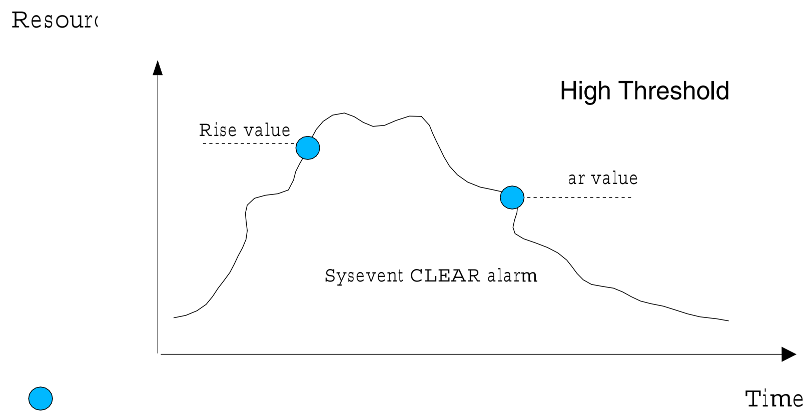

FIGURE 3–6 Rise and Clear Values for a High Threshold

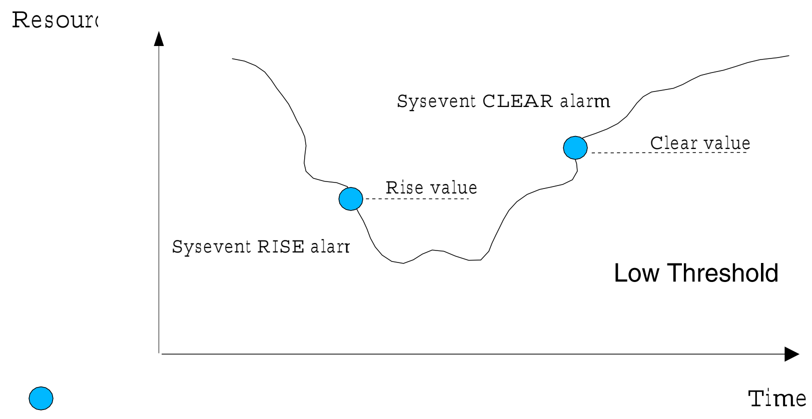

FIGURE 3–7 Rise and Clear Values for a Low Threshold

You can modify the value of the threshold rise and clear values dynamically. At system initialization time, the thresholds are disabled until they are set explicitly by an application.

In addition, the system exhibits a number of tunable values that you can modify dynamically to affect the behavior of the system. These values might, for example, represent the maximum number of open file descriptors per process, or IP forwarding behavior.

The values exposed are given symbolic names according to a tree schema, or they can be accessed through an object identifier (OID), obtained from the symbolic name of the value. The API for getting or setting, or getting and setting, these values is based on the `sysctl()` facility defined by FreeBSD systems. See the following section for details.

The `sysctl` Facility

The `sysctl` facility allows the retrieval of information from the system, and allows processes with appropriate privileges to set system information.

The information available from `sysctl` consists of integers, strings, tables, or opaque data structures. This information is organized in a tree structure containing two types of node:

Proxy leaf nodes	Access data acquired dynamically on demand. These nodes transparently handle the information exposed by the microkernel
Dynamically created nodes	Represent the information exposed by the devices, as it appears and disappears dynamically

Only proxy leaf nodes have data associated with them.

The `sysctl` nodes are natively identified using a management information base (MIB) style name, an OID, in the form of a unique array of integers.

`sysctl` API

Two `sysctl` system calls are provided:

Function	Description
`sysctl()`	Get/set a value identified by its OID
`sysctlbyname()`	Get/set a value identified by its name

For details, see the `sysctl`(1M) man page.

Device Instrumentation and Management

The `sysctl()` facility is used to expose the instrumentation information maintained by the device drivers. This information is retrieved via the Device Driver Manager (DDM).

The Device Driver Manager is a system component that enables a supervisor application to manage devices. Only the devices that export a management DDI interface or that have a parent that exports this DDI can be managed in this way. The DDM is an abstraction of the DKI and the management DDI.

The DDM is implemented as a set of functions that are organized in a library, and can only be used by one client at a time.

The DDM implements a tree of manageable devices with the following properties and features:

- A device can be in one of the following three run states: `DDM_RUNSTATE_ONLINE`, `DDM_RUNSTATE_OFFLINE`, and `DDM_RUNSTATE_INACTIVE`.
- A device can also be in one of the following availability states simultaneously: `DDM_AVSTATE_ENABLED` and `DDM_AVSTATE_DISABLED`.
- A device in an online state is able to audit its own health, and export some statistics (in addition to standard operation).
- A device in an offline state can only perform internal diagnostics
- A device in the inactive state does not perform any operations, although it is able to change its state to another value. One of the purposes of the shut-down state is to be able to change a property of the device in the device tree.
- A device in the `DDM_AVSTATE_ENABLED` state is able to have a driver running to manage it. However, a device in the `DDM_AVSTATE_DISABLED` state is locked and no drivers can be started to manage it.

Availability and run states are completely independent of each other, despite the fact that a disabled device may eventually be inactive.

The state of a device is changed on request from the `DDM` client or by external events, such as hardware failure or device hot swap. In both cases the DDM client is notified of the successful state change through a handler (callback) that is defined at the time of opening.

Device Tree

The initial internal device tree is built by taking all devices that satisfy the following criteria:

- All devices that export the `mngt` DDI.
- All devices that export the `diag` DDI.

- All devices that have a bus parent that exports the `mngt DDI`. This means that the child drivers can be shut down or initialized via their bus parent.

The tree of devices exposed by the DDM to its client is only a subset of the internal tree managed by the DDM. This in turn is a subset of the complete device tree for the current board. The way in which it is built is described in the preceding section.

The devices that are exposed via the DDM are:

- All devices that have a parent (so that they can be shut down or reinitialized).
- All diagnostic devices, as they are generally leaf devices, and not bus parent nodes.

The device tree API is summarized in the following table:

Function	Description
`svDdmAudit()`	Runs non-intrusive tests on an online device
`svDdmClose()`	Closes a previously made connection to the device driver manager
`svDdmDiag()`	Runs diagnostics on a node that is currently offline
`svDdmDisable()`	Locks the specified device node in the disabled state
`svDdmEnable()`	Enables a client to set the availability state of the specified device node to `DDM_AVSTATE_ENABLED`
`svDdmGetInfo()`	Enables the client of the DDM to obtain information on the specified node in the manageable device tree
`svDdmGetState()`	Enables the client of the DDM to get the state value of the specified node
`svDdmGetStats()`	Returns raw I/O statistics (counters) for an online device
`svDdmOffline()`	Enables the DDM client to set the run state of the specified node to `DDM_RUNSTATE_OFFLINE`
`svDdmOnline()`	Enables the DDM client to set the run state of the specified node to `DDM_RUNSTATE_ONLINE`
`svDdmOpen()`	Opens a connection to the device driver manager and obtains access to the management of the current device driver instances

Function	Description
`svDdmShutdown()`	Enables the DDM client to request that the driver running on the specified node is shut down

Related `sysctl()` entries

A number of `sysctl()` entries are present in the `sysctl` tree. Each device appears as a `sysctl` node that holds per-device information, under the top-level dev node. Available information about the device includes:

Name	Per-device information is stored in a `sysctl` node whose name derives from the canonical physical pathname of the device.
Class	This string holds the device class, if provided by the DDM. If no value is supplied, the content of this entry defaults to '`?`'.
Status	The integer contains both the availability and run status of the device, as provided by the DDM.
Statistics	This structure holds the device-class-specific statistics. Reading this node returns an error if the device does not export statistics.
Diagnostics	This entry triggers the diagnostic process of a device by writing a magic value to it (1), retrieves the result of the last diagnostic by reading it. An error may be returned if the device does not support diagnostics or if the diagnostics cannot run because the device is not in the appropriate state.
Audit	Similar to device diagnostics, this entry triggers the audit process and retrieves the result of the previous audit.

System Events

The `SYSTEM_EVENTS` feature enables a user-level application to be notified of the occurrence of events in the system and/or drivers. The following events are posted by the system and received by the application:

- Gauges crossing their threshold
- Creation or destruction of processes and, optionally, actors
- File system mounts and unmounts
- Detection of error in a driver
- Detection of error in the operating system

System events are carried by messages that are placed in different queues, depending upon the kind of events. In the ChorusOS operating system, the system events feature relies on the MIPC microkernel feature. The maximum number of system events that can be queued by the system is fixed by a tunable, set when you build the system.

The system events feature is also available to user-level applications to exchange events and is not restricted to system-level communication.

In the context of system events, the following terms are defined:

- An *event* is something that happens inside one entity corresponding to a change in the abstract state of that subsystem or application. Events are not generally observable from outside the entity, and cannot correspond to a change in the actual state of the entity. The entity in which the event occurs can notify certain user applications of the occurrence.
- An *event publisher* is the entity that notifies other entities of the occurrences of a particular set of events. Notification of occurrences of events can be made directly to interested entities or through an intermediary dispatcher. The events can be generic to a particular technology or specific to the event publisher.
- An *event subscriber* is an entity that is interested in the occurrence of certain events. It can subscribe its interest directly with the event publisher or with some intermediary entity to receive event notifications.
- An *event buffer* is passed from an event publisher to an event consumer to indicate that an event has occurred. The buffer includes information to describe the occurrence of an event in a particular publisher. The event buffer can be passed directly from the publisher to the consumer, or through an intermediary dispatcher.

At a minimum, an event is described by its event type, event identifier and publisher ID. These three fields combine to form the event buffer header. The goal is to provide a simple and flexible way to describe the occurrence of an event. If additional information is required to describe the event further, a publisher can provide a list of self-defined attributes. Event attributes contain an event attribute name/value pair that combine to define that attribute. Event attributes are used in event objects to provide self-defining data as part of the event buffer. The name of the event attribute is a character string. The event attribute value is a self-defining data structure that contains a data-type specifier and the appropriate union member to hold the value of the data specified.

Applications are provided a `libnvpair` to handle the attribute list and to provide a set of interfaces for manipulating name-value pairs. The operations supported by the library include adding and deleting name-value pairs, looking up values, and packing the list into contiguous memory to pass it to another address space. The packed and unpacked data formats are freshened internally. New data types and encoding methods can be added with backward compatibility.

To enable the code of this library to be linked to the Solaris kernel or to the ChorusOS operating system, the standard *errno* variable is not used to notify the caller that an error occurred. Error values are returned by the library functions directly.

System Events API

The system events API is summarized in the following table:

Function	Description
`sysevent_get_class_name()`	Get the class name of the event
`sysevent_get_subclass_name()`	Get the subclass name
`sysevent_get_size()`	Get the event buffer size
`sysevent_get_seq()`	Get the event buffer size
`sysevent_get_time()`	Get the time stamp
`sysevent_free()`	Free memory for system event handle
`sysevent_post_event()`	Post a system event from userland
`sysevent_get_event()`	Wait for a system event
`sysevent_get_attr_list()`	Get the attribute list pointer
`sysevent_get_vendor_name()`	Get the publisher vendor name
`sysevent_get_pub_name()`	Get the publisher name
`sysevent_get_pid()`	Get the publisher PID
`sysevent_lookup_attr()`	Search the attribute list
`sysevent_attr_next()`	Returns the next attribute associated with event
`sysevent_dup()`	Duplicate a system event

OS_GAUGES

The `OS_GAUGES` module generates system events related to the OS component of the ChorusOS operating system, following alarms or signals generated by gauges, counters and thresholds. These system events are passed to the `C_OS`.

The `OS_GAUGES` module has no dedicated system calls, but rather reads and controls counters, gauges and thresholds through `sysctl()`, `sysctlbyname()`, and the `/PROCFS` file system.

For details, see the `INSTRUMENTATION`(5FEA) man page.

Microkernel Statistics (`MKSTAT`)

Statistics regarding the microkernel are provided to the `C_OS` by the `MKSTAT` module. Statistics for events such as alarms and creation or deletion of ChorusOS actors and POSIX processes are retrieved by `sysctl` and `/proc` and then grouped by function type in the `MKSTAT` module.

For details, see the `INSTRUMENTATION`(5FEA) man page.

`MKSTAT` API

The `MKSTAT` API is summarized in the following table:

Function	Description
`mkStatMem()`	Memory statistics
`mkStatSvPages()`	Supervisor page statistics
`mkStatActors()`	mkStatThreads
`mkStatThreads()`	Execution statistics
`mkStatCpu()`	CPU statistics
`mkStatActorMem()`	Per-actor statistics
`mkStatActorSvPages()`	Supervisor per-actor statistics
`mkStatThreadCpu()`	Per-thread statistics
`mkStatEvtCtrl()`	Event control statistics
`mkStatEvtWait()`	Events waiting statistics

Microkernel Memory Instrumentation

The `C_OS` implements the microkernel memory instrumentation via the `sysctl kern.mkstats.mem` node. The `OS_GAUGES` feature must be set to `true`.

Instrumentation related to memory use comprises the following measurements:

Function	Instrument Type	Description
`physPagesEquiped()`	Attribute	Measures the amount of physical pages of memory available on the node

Function	Instrument Type	Description
`physPagesavail()`	Gauge (low threshold)	Measures the amount of physical pages of memory currently available
`allocFailures()`	Counter	Number of memory allocation failures since boot
`pageSize()`	Attribute	Size in bytes of physical page

Microkernel Supervisor Page Instrumentation

The `C_OS` implements the microkernel supervisor page instrumentation via the `sysctl` `kern.mkstats.svpages` node. The `OS_GAUGES` feature must be set to `true`.

Instrumentation related to use of supervisor pages comprises the following measurement:

Function	Instrument Type	Description
`svPages()`	Gauge (high threshold)	Measures number of supervisor pages currently allocated

Microkernel Execution Instrumentation

The `C_OS` implements the microkernel execution instrumentation via the `sysctl` `kern.mkstats.actors` and `kern.mkstats.threads` nodes. The `OS_GAUGES` feature must be set to `true`.

Instrumentation related to microkernel execution function comprises the following measurements:

Function	Instrument Type	Description
`maxActors()`	Attribute	Measures the maximum number of actors that can be created
`actors()`	Gauge (high threshold)	Measures the current number of actors in use

Function	Instrument Type	Description
`maxThreads()`	Attribute	Measures the maximum number of threads that can be created
`threads()`	Gauge (high threshold)	Measures the current number of threads in use

Microkernel CPU Instrumentation

The `C_OS` implements the microkernel CPU instrumentation via the `sysctl kern.mkstats.cpu` node.

Instrumentation related to microkernel CPU use comprises the following measurements:

Function	Instrument Type	Description
`total_cpu()`	Counter	Measures the number of milliseconds CPU has been used since boot
`external()`	Counter	Measures the number of milliseconds the CPU has been used outside execution actor since boot (similar to UNIX supervisor mode)
`internal()`	Counter	Measures the number of milliseconds the CPU has been used inside execution actor supervisor mode since boot (similar to UNIX user mode)

This basic instrumentation provides only raw measurements on top of which applications can compute ratios of CPU use according to their needs.

POSIX Process Instrumentation

The `C_OS` implements the microkernel POSIX process instrumentation via the `sysctl kern.mkstats.procs` node.

Instrumentation related to microkernel processes comprises the following measurements:

Function	Instrumentation Type	Description
`procs()`	Gauge (high threshold)	Measures the current number of processes in use on the node
`nb_syscalls()`	Counter	Counts the number of system calls performed since boot
`nb_syscalls_failures()`	Counter	Counts the number of failed system calls since boot
`nb_fork_failures()`	Counter	Counts the number of failed `fork()` system calls since boot

File Instrumentation

The `C_OS` implements the microkernel file instrumentation via the `sysctl kern.mkstats.files` node.

Instrumentation related to microkernel file use comprises the following measurements:

Function	Instrument Type	Description
`open_files()`	Gauge (high threshold)	Measures the current number of open files
`vnodes()`	Gauge (high threshold)	Current number of used virtual nodes (`vnodes`)

Per-File System Instrumentation

The following instrumentation is available for each mounted file system:

Function	Instrument Type	Description
`fs_status()`	Attribute	Determines availability of threshold controls (for example, a read-only mounted file system has no threshold control)
`fs_max_size()`	Attribute	Size of the file system in blocks
`fs_bsize()`	Attribute	Size in bytes of the block
`fs_space_free()`	Gauge (low threshold)	Number of blocks currently available in the file system
`fs_max_files()`	Attribute	Maximum number of files that can be created on the file system
`fs_nb_files()`	Gauge	Current number of files created on the file system

Per-Actor and Per-Process Instrumentation

For each actor or process currently active on the system, the following information is available to the `C_OS` via the `stats` entry of the process directory in the `/proc` file system:

Function	Instrument Type	Description
`virtpages()`	Gauge (high threshold)	Counts the number of virtual memory pages used by an actor
`physPages()`	Simple Gauge	Counts the number of physical memory pages used by an actor
`lockPages()`	Simple Gauge	Number of locked memory pages used by an actor
`process_virt_pages()`	Gauge (high threshold)	Number of virtual memory pages used by a process

Function	Instrument Type	Description
`process_phys_pages()`	Simple Gauge	Number of physical memory pages used by a process
`process_lock_pages()`	Simple Gauge	Number of locked memory pages used by a process
`open_files()`	Gauge (high threshold)	Current number of open file descriptors
`internal_cpu()`	Counter	Cumulated (all threads) internal CPU usage in milliseconds (similar to user mode)
`external_cpu()`	Counter	Cumulated (all threads) external CPU usage in milliseconds (similar to system mode)

Microkernel Per-Thread Instrumentation

For each thread currently active on the system, the following information is available via the `stats` entry of the process directory in the `/proc` file system:

Function	Instrument Type	Description
`internal_cpu()`	Counter	Internal CPU time spent in milliseconds (similar to user mode)
`external_cpu()`	Counter	External CPU time spent in milliseconds (similar to supervisor mode)
`waiting_cnt()`	Counter	Number of times the thread has been blocked

Optional Java Functionality

The ChorusOS operating system offers the following optional Java functionalities.

Java Runtime Environment (JRE)

The ChorusOS Java Runtime Environment (JRE) component allows you to develop and implement Java applications on the ChorusOS operating system. The ChorusOS Java Runtime Environment provides the following services:

Java 2 Platform Micro Edition (J2ME) Compatibility

The ChorusOS JRE offers conformity with the Java 2 Platform Micro Edition (J2ME) specification, and meets the criteria of the Java 2 Technology Conformance Kit (TCK). It supports the APIs for J2ME Connected Device Configuration (CDC) and the Foundation profile. The pre-FCS RMI profile can also be used with source deliveries.

C Virtual Machine (CVM)

A C virtual machine (CVM) allows applications written in the Java programming language to be portable across different hardware environments and operating systems. The CVM mediates between the application and the underlying platform, converting the application's bytecodes into machine-level code appropriate for the hardware and the ChorusOS operating system. The CVM supports all ChorusOS CPUs and it uses native ChorusOS threads with tunable priority levels. It is possible for several CVMs to run simultaneously.

The ChorusOS CVM offers the following characteristics:

- The CVM and user Java applications can be launched directly from an image, embedded in flash or read-only memory. The CVM also offers Execute-in-Place (XIP) functionality, reducing the size of the footprint of your application.
- A Java Native Interface (JNI).
- The CVM uses a generational garbage collector (GC), and supports the fastest CVM locking mode, using atomic operations.

Java Platform Debugger Architecture (JPDA)

The ChorusOS JRE provides debugging support via the Java Platform Debugger Architecture (JPDA). JPDA provides the infrastructure needed to build end-user debugger applications. JPDA consists of the layered APIs:

Java Debug Interface (JDI)
: A high-level Java programming language interface, including support for remote debugging.

Java Debug Wire Protocol (JDWP):
: Defines the format of information and requests transferred between the debugging process and the debugger front-end.

Java Virtual Machine Debug Interface (JVDMI):
: A low-level native interface. Defines the services a Java virtual machine must provide for debugging.

The Sun Forte™ for Java debugger fully supports JPDA.

Java Dynamic Management Kit (JDMK)

The ChorusOS operating system supports the Java Dynamic Management Kit (JDMK).

JDMK allows you to develop Java technology-based agents on your platform. These agents can access your resources through the Java Native Interface or you can take advantage of the Java programming language to develop new resources in the Java Dynamic Management agent.

The Java Dynamic Management Kit provides scheduling, monitoring, notifications, class loading, and other agent-side services. Agents running in the CVM are completely scalable, meaning that both resources and services may be added or removed dynamically, depending on platform constraints and run-time needs. Connectors and protocol adaptors let you develop Java technology-based management applications that may access and control any number of agents transparently through protocols such as RMI, HTTP, SNMP, and HTML.

Tools

The ChorusOS operating system provides the following tools:

Ews Graphic Configuration Tool

The ChorusOS operating system offers a graphic configuration tool, called Ews, to help you configure your system. The Ews configuration tool allows you to:

- Configure the system image, to include or exclude different features and components
- Configure the features and tunables in your system image
- Set the environment variables

- Add actors to the system image.

For details about using the Ews graphic configuration tool, see the *ChorusOS 5.0 Application Developer's Guide*.

Built-in Debugging Tools

The ChorusOS operating system provides embedded debugging tools that debug all parts of the operating system, including the boot.

Debugging Architecture

The ChorusOS operating system includes an open debugging architecture, as specified by the *ChorusOS 5.0 Debugging Guide*. The debug architecture relies on a host-resident server which abstracts the target platform to host tools, in particular debuggers.

The debug server is intended to connect to various forms of target systems, through connections such as target through serial line or target through Ethernet.

This debug architecture provides support for two debugging modes:

- Application debug
- System debug

In the application debugging mode, debuggers connect to multi-threaded processes or actors. Debugging an actor is non-intrusive for the system and other actors, except for actors expecting services from the actor.

In system debugging mode, debuggers connect to the operating system seen as a virtual single multi-threaded process. Debugging the system is highly intrusive, since a breakpoint will stop all system operations. System debugging is designed to allow debugging of all the various parts of the operating system, for example: the boot sequence, the microkernel, the BSP and the system protocol stacks.

Tools support

The ChorusOS operating system provides the following features to support debugging.

LOG

The LOG feature provides support for logging console activity on a target system.

For details, see sysLog(2K).

PERF

The PERF feature provides an API to share the system timer (clock) in two modes:

- A free-running mode, which causes the timer to overflow after reaching its maximum value and continue to count up from its minimum value. This mode can be used for fine-grained execution measurement. This deactivates the system clock.
- A periodic mode, where the system timer is shared between the application and the system tick. The timer will generate an interrupt at a set interval. The application handler will be invoked at the required period. This mode can be used by applications such as profilers.

The PERF API closely follows the timer(9DDI) device driver interface.

For details, see PERF(5FEA).

MON

The MON feature provides a means to monitor the activity of microkernel objects such as threads, actors, and ports. Handlers can be connected to the events related to these objects so that, for example, information related to thread-sleep/wake events can be known. Handlers can also monitor global events, affecting the entire system.

For details, see MON(5FEA).

SYSTEM_DUMP

The ChorusOS operating system dump feature is also used for debugging the system in the event of a crash. See "System Dump (SYSTEM_DUMP)" on page 126 for details.

DEBUG_SYSTEM

The DEBUG_SYSTEM feature enables remote debugging with the GDB Debugger for the ChorusOS operating system. GDB communicates with the ChorusOS debug server (see chserver(1CC)) through the RDBD protocol adapter (see rdbd(1CC)), both running on the host. The debug server in turn communicates with the debug agent running on the target. The debug server exports an open Debug API, which is documented and available for use by third party tools.

For details, see DEBUG_SYSTEM(5FEA).

APPENDIX **A**

Optional ChorusOS Operating System Components

This Appendix lists the optional features of the ChorusOS operating system, broken down according to function.

The following table shows the optional component groups.

TABLE A–1 Optional Operating System Components

Component	Name
Actor management	
User-mode extension support	`USER_MODE`
Dynamic libraries	`DYNAMIC_LIB`
Compressed file management	`GZ_FILE`
Scheduling	
FIFO scheduling	`SCHED_FIFO`
Multi-class scheduling	`SCHED_CLASS`
Round robin scheduling class	`SCHED_CLASS_RR`
Real-time scheduling class	`SCHED_CLASS_RT`
Memory management	
Virtual (user and supervisor) address space	`VIRTUAL_ADDRESS_SPACE`
On-demand paging	`ON_DEMAND_PAGING`
System Instrumentation	
Microkernel statistics	`MKSTAT`
Solaris system events	`SOLARIS_SYSEVENTS`

TABLE A–1 Optional Operating System Components *(Continued)*

Component	Name
Operating system gauges	`OS_GAUGES`
High Availability	
Hot restart	`HOT_RESTART`
Watchdog timer	`WDT`
Black box	`BLACKBOX`
System dump	`SYSTEM_DUMP`
Inter-thread synchronization	
Semaphores	`SEM`
Event flag sets	`EVENT`
Mutexes	`MUTEX`
Mutual exclusion locks supporting thread priority inversion avoidance	`RTMUTEX`
Management	
Periodic timers	`TIMER`
Thread and actor virtual timer	`VTIMER`
Date and time of day	`DATE`
Real-time clock	`RTC`
Environment variables	`ENV`
Inter-thread communication	
Location-transparent inter-process communication	`IPC`
Remote (inter-site) IPC support	`IPC_REMOTE`
Mailbox-based communications mechanism	`MIPC`
Inter-thread synchronization	`MONITOR`
POSIX I/O system calls	`POSIX-FILEIO`
POSIX semaphores	`POSIX-SEM`
POSIX sockets	`POSIX_SOCKETS`
POSIX threads	`POSIX-THREADS`
POSIX timers	`POSIX-TIMERS`
POSIX message queues	`POSIX_MQ`

TABLE A–1 Optional Operating System Components *(Continued)*

Component	Name
POSIX shared memory objects	`POSIX_SHM`
POSIX real-time signals	`POSIX_REALTIME_SIGNALS`
Private per-thread data	`PRIVATE-DATA`
Local name server for LAP binding	`LAPBIND`
LAP validity-check option	`LAPSAFE`
Tools support	
System logging	`SYSLOG`
Message logging	`LOG`
Profiling and benchmark support	`PERF`
System monitoring	`MON`
System debugging	`DEBUG_SYSTEM`
Core dump	`CORE_DUMP`
C_INIT	
Basic command interpreter on target	`LOCAL_CONSOLE`
Remote shell	`RSH`
File system options	
Named pipes	`FIFOFS`
NFS client	`NFS_CLIENT`
NFS server	`NFS_SERVER`
MS-DOS file system	`MSDOSFS`
PDE file system	`PDEVFS`
`/proc` file system	`PROCFS`
UFS file system	`UFS`
ISO9000 file system	`ISOFS`
I/O management	
`/dev/mem`, `/dev/kmem`, `/dev/zero`	`DEV_MEM`
Support for RAM disk	`RAM_DISK`
Chorus Mapper (supports virtual memory only)	FS_MAPPER

TABLE A–1 Optional Operating System Components *(Continued)*

Component	Name
Support for FLASH media	`FLASH`
Access raw memory device	`RAWFLASH`
Virtual TTY	`VTTY`
Driver for SCSI disk	`SCSI_DISK`
NVRAM device	`DEV_NVRAM`
CD-ROM device	`DEV_CDROM`
Networking	
POSIX 1003.1g-compliant sockets	`POSIX_SOCKETS`
Point-to-point protocols	`PPP`
Local sockets and pipes (used by `POSIX_SOCKETS`)	`AF_LOCAL`
Socket routing (used by `POSIX_SOCKETS`)	`AF_ROUTE`
IPv4 sockets (used by `POSIX_SOCKETS`)	`AF_INET`
IPv6 sockets (used by the `IPV6` feature)	`AF_INET6`
Internet Protocol version 6	`IPV6`
Berkley Packet Filter	`BPF`
Support for OSI	`IOM_OSI`
Support for IPC	`IOM_IPC`
Network Time Protocol	`NTP`

APPENDIX **B**

Complete List of Available ChorusOS System Calls

Below is the complete list of system calls available in the ChorusOS operating system.

POSIX APIs

POSIX System Calls (2POSIX)

accept	access	adjtime
bind	chdir	chflags
chmod	chown	chroot
close	connect	dup
dup2	fchdir	fchflags
fchmod	fchown	fcntl
flock	fpathconf	fstat
fstatfs	fsync	ftruncate
getdirentries	getdomainname	getegid
geteuid	getfh	getfsstat
getgid	gethostname	getpeername
getpgid	getpgrp	getpid

getppid	getrlimit	getsid
getsockname	getsockopt	gettimeofday
getuid	hostname	ioctl
issetugid	kill	link
listen	lseek	lstat
mkdir	mkfifo	mknod
mmap	mount	mq_close
mq_getattr	mq_open	mq_receive
mq_send	mq_setattr	mq_unlink
munmap	nfssvc	nvramapi
open	pathconf	pipe
poll	posix_spawn	posix_spawnp
read	readlink	readv
recv	recvfrom	recvmsg
rename	rmdir	select
send	sendmsg	sendto
setdomainname	setegid	seteuid
setgid	sethostname	setpgid
setpgrp	setrlimit	setsid
setsockopt	settimeofday	setuid
shm_open	shm_unlink	shutdown
sigaction	sigpending	sigprocmask
sigqueue	sigsuspend	sigtimedwait
sigwait	sigwaitinfo	socket
socketpair	stat	statfs
swapon	symlink	sync
truncate	umask	unlink
unmount	utimes	wait
write	writev	

FTP Daemon Library (3FTPD)

ftpdGetCnx	ftpdHandleCnx	ftpdOob
ftpdStartSrv	lreply	perror_reply
reply	systemAsciiOff	systemBeany
systemBesuper	systemBeuser	systemChdir
systemCommand	systemDelete	systemFileSize
systemGunique	systemListFiles	systemLog
systemLogwtmp	systemMkdir	systemPass
systemReceiveAscii	systemReceiveBin	systemRename
systemRmdir	systemSendAscii	systemSendBin
systemSetThreadTitle	systemSleep	systemUser
systemVlog		

Mathematical Libraries (3M)

acos	acosh	asin
asinh	atan	atan2
atanh	cabs	cbrt
ceil	copysign	cos
cosh	drem	erf
erfc	exp	expm1
finite	floor	fmod
gamma	hypot	ieee
infnan	j0	j1
jn	lgamma	log
log10	log1p	logb
pow	rint	scalb
sin	sinh	sqrt

tan	tanh	y0
y1	yn	

POSIX Library Functions (3POSIX)

Clocks and Timers		
clock_gettime	clock_settime	clock_getres
timer_create	timer_delete	timer_getoverrun
timer_gettime	timer_settime	nanosleep

Threads	
pthread_attr_destroy	pthread_attr_getdetachstate
pthread_attr_getinheritsched	pthread_attr_getschedparam
pthread_attr_getschedpolicy	pthread_attr_getscope
pthread_attr_getstackaddr	pthread_attr_getstacksize
pthread_attr_init	pthread_attr_setdetachstate
pthread_attr_setinheritsched	pthread_attr_setschedparam
pthread_attr_setschedpolicy	pthread_attr_setscope
pthread_attr_setstackaddr	pthread_attr_setstacksize
pthread_cond_broadcast	pthread_cond_destroy
pthread_cond_init	pthread_cond_signal
pthread_cond_timedwait	pthread_cond_wait
pthread_condattr_destroy	pthread_condattr_init
pthread_create	pthread_equal
pthread_exit	pthread_getschedparam
pthread_getspecific	pthread_join
pthread_key_create	pthread_key_delete
pthread_kill	pthread_mutex_destroy
pthread_mutex_init	pthread_mutex_lock

Threads	
pthread_mutex_trylock	pthread_mutex_unlock
pthread_mutexattr_destroy	pthread_mutexattr_init
pthread_self	pthread_setschedparam
pthread_setspecific	pthread_yield
pthread_once	

Baud RateFunctions	
cfgetispeed	cfgetospeed
cfmakeraw	cfsetispeed
cfsetospeed	cfsetspeed

Terminal Interface Control	
tcgetattr	tcsetattr

Execution Scheduling	
sched_get_priority_max	sched_get_priority_min
sched_rr_get_interval	sched_yield

Synchronization	
sem_destroy	sem_getvalue
sem_init	sem_post
sem_trywait	sem_wait

Other POSIX Library Functions	
err	closedir
directory	getcwd
getwd	opendir
readdir	rewinddir
seekdir	sysconf

Other POSIX Library Functions	
sysctl	sysctlbyname
telldir	

Other	
btree	cancellation
dbopen	endnetent
endnetgrent	endprotoent
endservent	getdiskbyname
getmntinfo	getnetbyaddr
getnetbyname	getnetent
getnetgrent	getprotobyname
getprotobynumber	getprotoent
getservbyname	getservbyport
getservent	glob
globfree	hash
innetgr	link_addr
link_ntoa	mpool
ns_addr	ns_ntoa
	recno
setnetent	setnetgrent
setprotoent	setservent
verr	verrx
vwarn	vwarnx
warn	warnx

RPC Services (3RPC)

auth_destroy	authnone_create

authsys_create	authsys_create_default
clnt_call	clnt_control
clnt_create	clnt_create_timed
clnt_create_vers	clnt_create_vers_timed
clnt_destroy	clnt_dg_create
clnt_freeres	clnt_geterr
clnt_pcreateerror	clnt_perrno
clnt_perror	clnt_raw_create
clnt_send	clnt_spcreateerror
clnt_sperrno	clnt_sperror
clnt_tli_create	clnt_tp_create
clnt_tp_create_timed	clnt_vc_create
endnetpath	endrpcent
getnetpath	getrpcbyname
getrpcbynumber	getrpcent
getrpcport	rpc
rpc_broadcast	rpc_broadcast_exp
rpc_call	rpc_clnt_auth
rpc_clnt_calls	rpc_clnt_create
rpc_control	rpc_createerr
rpc_reg	rpc_svc_calls
rpc_svc_create	rpc_svc_err
rpc_svc_reg	rpc_xdr
rpcb_getaddr	rpcb_getmaps
rpcb_gettime	rpcb_rmtcall
rpcb_set	rpcb_unset
rpcbind	setnetpath
setrpcent	svc_add_input
svc_auth_reg	svc_control
svc_create	svc_destroy

svc_dg_create	svc_dg_enablecache
svc_done	svc_exit
svc_fd_create	svc_fdset
svc_freeargs	svc_getargs
svc_getreq_common	svc_getreq_poll
svc_getreqset	svc_getrpccaller
svc_max_pollfd	svc_pollfd
svc_raw_create	svc_reg
svc_run	svc_sendreply
svc_tli_create	svc_tp_create
svc_unreg	svc_vc_create
svcerr_auth	svcerr_decode
svcerr_noproc	svcerr_noprog
svcerr_progvers	svcerr_systemerr
svcerr_weakauth	xdr
xdr_accepted_reply	xdr_admin
xdr_array	xdr_authsys_parms
xdr_bool	xdr_bytes
xdr_callhdr	xdr_callmsg
xdr_char	xdr_complex
xdr_control	xdr_create
xdr_destroy	xdr_double
xdr_enum	xdr_float
xdr_free	xdr_getpos
xdr_hyper	xdr_inline
xdr_int	xdr_long
xdr_longlong_t	xdr_opaque
xdr_opaque_auth	xdr_pointer
xdr_quadruple	xdr_reference
xdr_rejected_reply	xdr_replymsg

xdr_setpos	xdr_short
xdr_simple	xdr_sizeof
xdr_string	xdr_u_char
xdr_u_hyper	xdr_u_int
xdr_u_long	xdr_u_longlong_t
xdr_u_short	xdr_union
xdr_vector	xdr_void
xdr_wrapstring	xdrmem_create
xdrrec_create	xdrrec_endofrecord
xdrrec_eof	xdrrec_readbytes
xdrrec_skiprecord	xdrstdio_create
xprt_register	xprt_unregister

Standard C Library Functions (3STDC)

These services are available to applications using the POSIX subsystem.

_assert	_ldexp	_stdc_assert
abort	abs	addr2ascii
alarm	alphasort	ascii2addr
asctime	asctime_r	assert
atexit	atof	atoi
atol	bcmp	bcopy
bsearch	bstring	byteorder
bzero	calloc	cgetcap
cgetclose	cgetent	cgetfirst
cgetmatch	cgetnext	cgetnum
cgetset	cgetstr	cgetustr
clearerr	closelog	crypt
ctime	ctime_r	ctype

difftime	div	dn_comp
dn_expand	endgrent	endhostent
endnetconfig	endpwent	errno
ether_aton	ether_hostton	ether_line
ether_ntoa	ether_ntohost	ethers
exit	fabs	fclose
fdopen	feof	ferror
fflagstostr	fflush	ffs
fgetc	fgetpos	fgets
fileno	flockfile	fopen
fprintf	fputc	fputs
fread	free	freenetconfigent
freopen	fscanf	fseek
fsetpos	ftell	ftrylockfile
funlockfile	fwrite	gai_unlocked
getaddrinfo	getbsize	getc
getc_unlocked	getchar	getchar_unlocked
getenv	getgrent	getgrid
getgrnam	gethostbyaddr	gethostbyname
gethostbyname2	gethostent	getnetconfig
getnetconfigent	getopt	getpwent
getpwname	getpwuid	gets
getsitebyaddr	getsitebyname	getsubopt
getw	gmtime	gmtime_r
herror	htonl	htons
if_freenameindex	if_indextoname	if_nameindex
if_nametoindex	index	inet
inet6_option_alloc	inet6_option_append	inet6_option_find
inet6_option_init	inet6_option_next	inet6_option_space
inet6_rthdr_add	inet6_rthdr_getaddr	inet6_rthdr_getflags

inet6_rthdr_init	inet6_rthdr_lasthop	inet6_rthdr_reverse
inet6_rthdr_segments	inet6_rthdr_space	inet_addr
inet_aton	inet_lnaof	inet_makeaddr
inet_netof	inet_network	inet_ntoa
inet_ntop	inet_pton	initstate
isalnum	isalpha	isascii
isatty	iscntrl	isdigit
isgraph	isinf	islower
isnan	isprint	ispunct
isspace	isupper	isxdigit
killpg	labs	ldexp
ldiv	localtime	localtime_r
longjmp	malloc	memccpy
memchr	memcmp	memcpy
memmove	memory	memset
mkstemp	mktemp	mktime
modf	nc_perror	nc_sperror
netdir	netdir_free	netdir_getbyaddr
netdir_getbyname	netdir_mergeaddr	netdir_options
netdir_perror	netdir_sperror	ntohl
ntohs	openlog	pause
pclose	perror	popen
printerr	printf	putc
putc_unlocked	putchar	putchar_unlocked
putenv	puts	putw
qsort	rand	rand_r
random	realloc	realpath
regcomp	regerror	regex
regexec	regfree	remove
res_init	res_mquery	res_query

res_search	res_send	resolver
rewind	rindex	scandir
scanf	setbuf	setenv
setgrent	setgroupent	sethostent
setjmp	setlogmask	setnetconfig
setpassent	setpwent	setstate
setvbuf	snprintf	sprintf
srand	srandom	sscanf
stdarg	strcasecmp	strcat
strchr	strcmp	strcoll
strcpy	strcspn	strdup
strerror	strftime	string
strlen	strncasecmp	strncat
strncmp	strncpy	strpbrk
strrchr	strsep	strspn
strstr	strtod	strtofflags
strtok	strtok_r	strtol
strtoul	strxfrm	swab
sys_errlist	sys_nerr	syslog
taddr2uaddr	tempnam	thread_once
time	tmpfile	tmpnam
toascii	tolower	toupper
tzset	uaddr2taddr	ungetc
unlocked	unsetenv	vfprintf
vprintf	vsnprintf	vsprintf
vsyslog		

Telnet Services (3TELD)

inetAccept	inetBind	inetClient
inetClose	telnetdFlush	telnetdFree
telnetdGetTermState	telnetdInit	telnetdPasswd
telnetdRead	telnetdReadLine	telnetdSetTermState
telnetdUser	telnetdWrite	

Legacy POSIX-Like Extended APIs

These APIs are provided for backward compatibility only. To facilitate the migration of ChorusOS microkernel applications to POSIX, the POSIX equivalent, if applicable, is indicated below. A "-" is used to indicate that no equivalent service is provided.

Pre-POSIX service	POSIX equivalent service
acap	-
aconf	sysconf
acreate	-
acred	getuid, geteuid, getgid, getegid, setuid, setgid, seteuid, setegid
afexec	posix_spawn
afexecl	execl
afexecle	execle
afexeclp	execlp
afexecv	execv
afexecve	execve
afexecvp	execvp
agetalparam	-
agetId	getpid, getppid
akill	kill

Pre-POSIX service	POSIX equivalent service
aload	-
alParamBuild	-
alParamUnpack	-
astart	-
astat	pstat
atrace	ptrace
await	waitpid
awaits	wait
dladdr	dladdr
dlclose	dlclose
dlerror	dlerror
dlopen	dlopen
dlsym	dlsym

New POSIX-Like Extended APIs

POSIX System Calls (2POSIX)

creat	execve	fork
getegid	geteuid	getgid
getgroups	getpid	getppid
getuid	kill	nvramapi
setegid	seteuid	setgid
setuid	sigaction	sigpending
sigprocmask	sigqueue	sigsuspend
sigtimedwait	sigwait	sigwaitinfo

wait	waitpid

POSIX Library Functions (3POSIX)

alarm	exec	execl
execle	execlp	execv
execvp	fnmatch	pause
pthread_cancel	pthread_setcancelstate	pthread_setcanceltype
pthread_testcancel	pthread_cleanup_push	pthread_cleanup_pop
ptrace	sem_close	sem_open
sem_unlink	sigaddset	sigdelset
sigemptyset	sigfillset	sigismember
sigsetops	sleep	times
uname	utime	

System Microkernel APIs

The microkernel APIs include the services listed below. For convenience, these services have been divided into the corresponding man page sections.

Microkernel System Calls (2K)

_exit	actorCreate
actorDelete	actorName
actorPi	actorPrivilege
actorSelf	actorStart
actorStat	actorStop
bbClose	bbEvent

bbFilters	bbFreeze
bbGetNbb	bbList
bbName	bbProdis
bbRead	bbRelease
bbReset	bbSeverity
ethIpcStackAttach	ethOsiStackAttach
eventClear	eventInit
eventPost	eventWait
grpAllocate	grpPortInsert
grpPortRemove	ipcCall
ipcGetData	ipcReceive
ipcReply	ipcRestore
ipcSave	ipcSend
ipcSysInfo	ipcTarget
lapDescDup	lapDescIsZero
lapDescZero	lapInvoke
lapResolve	mkStatActorMem
mkStatActorSvPages	mkStatActors
mkStatCpu	mkStatEvtCtrl
mkStatEvtWait	mkStatMem
mkStatSvPages	mkStatThreadCpu
mkStatThreads	monitor
monitorGet	monitorInit
monitorNotify	monitorNotifyAll
monitorRel	monitorWait
msgAllocate	msgFree
msgGet	msgPoolStat
msgPut	msgQueueStat
msgRemove	msgSpaceCreate
msgSpaceOpen	mutexGet

mutexInit	mutexRel
mutexTry	padGet
padKeyCreate	padKeyDelete
padSet	portCreate
portDeclare	portDelete
portDisable	portEnable
portGetSeqNum	portLi
portMigrate	portPi
portUi	ptdErrnoAddr
ptdGet	ptdKeyCreate
ptdKeyDelete	ptdRemoteGet
ptdRemoteSet	ptdSet
ptdThreadDelete	ptdThreadId
rgnAllocate	rgnDup
rgnFree	rgnInitFromActor
rgnMapFromActor	rgnSetInherit
rgnSetOpaque	rgnSetPaging
rgnSetProtect	rgnStat
rtMutexGet	rtMutexInit
rtMutexRel	rtMutexTry
schedAdmin	semInit
semP	semV
svAbortHandler	svActorAbortHandler
svActorAbortHandlerConnect	svActorAbortHandlerDisconnect
svActorAbortHandlerGetConnected	svActorExcHandler
svActorExcHandlerConnect	svActorExcHandlerDisconnect
svActorExcHandlerGetConnected	svActorStopHandler
svActorStopHandlerConnect	svActorStopHandlerDisconnect
svActorStopHandlerGetConnected	svActorVirtualTimeout
svActorVirtualTimeoutCancel	svActorVirtualTimeoutSet

svCopyIn	svCopyInString
svCopyOut	svDdmAudit
svDdmClose	svDdmDiag
svDdmDisable	svDdmEnable
svDdmGetInfo	svDdmGetState
svDdmGetStats	svDdmOffline
svDdmOnline	svDmOpen
svDdmShutdown	svExcHandler
svFpuContext	svGetInvoker
svIntrLevel	svLapBind
svLapCreate	svLapDelete
svLapUnbind	svMaskedLockGet
svMaskedLockInit	svMaskedLockRel
svMaskedLockTry	svMemRead
svMemWrite	svMsgHandler
svMsgHdlReply	svPagesAllocate
svPagesFree	svPreemptable
svSoftIntrDeclare	svSoftIntrForget
svSoftIntrTrigger	svSpinLockGet
svSpinLockInit	svSpinLockRel
svSpinLockTry	svSysCtx
svSysPanic	svSysTimeout
svSysTimeoutCancel	svSysTimeoutSet
svSysTrapHandler	svSysTrapHandlerConnect
svSysTrapHandlerDisconnect	svSysTrapHandlerGetConnected
svThreadVirtualTimeout	svThreadVirtualTimeoutCancel
svThreadVirtualTimeoutSet	svTimeoutGetRes
svTrapConnect	svTrapDisConnect
svVirtualTimeoutCancel	svVirtualTimeoutSet
sysBench	sysGetConf

sysGetEnv	sysLog
sysPoll	sysRead
sysReboot	sysSetEnv
sysShutdown	sysTime
sysTimeGetRes	sysTimer
sysTimerGetCounterFrequency	sysTimerGetCounterPeriod
sysTimerReadCounter	sysTimerStartFreerun
sysTimerStartPeriodic	sysTimerStop
sysUnsetEnv	sysWrite
threadAbort	threadAborted
threadActivate	threadBind
threadContext	threadCreate
threadDelay	threadDelete
threadLoadR	threadName
threadResume	threadScheduler
threadSelf	threadSemInit
threadSemPost	threadSemWait
threadStart	threadStat
threadStop	threadStoreR
threadSuspend	threadTimes
timerCreate	timerDelete
timerGetRes	timerSet
timerThreadPoolInit	timerThreadPoolWait
uiBuild	uiClear
uiEqual	uiGetSite
uiIsLocal	uiLocalSite
uiSiteBuild	uiValid
univTime	univTimeAdjust
univTimeSet	virtualTimeGetRes
vmCopy	vmFree

vmLock	vmPageSize
vmPhysAddr	vmSetPar
vmStat	vmUnLock
wdt_get_interval	wdt_alloc
wdt_arm	wdt_disarm
wdt_free	wdt_get_maxinterval
wdt_get_mininterval	wdt_is_armed
wdt_pat	wdt_realloc
wdt_set_interval	wdt_shutdown
wdt_startup_commit	

Data Link Services (2DL)

svDataLink	svDataLinkAttach
svInputFrameDeliver	svLinkFailure
svOutFrameFree	

Monitoring Services (2MON)

KcModule	svActorMonConst
svActorPortList	svActorProbeConnect
svActorProbeDisconnect	svActorThreadList
svPortMonConst	svPortProbeConnect
svPortProbeDisconnect	svSiteActorList
svSiteMonConst	svSiteProbeConnect
svSiteProbeDisconnect	svThreadMonConst
svThreadProbeConnect	svThreadProbeDisconnect
threadMonUser	UcModule

Virtual Memory Segment Services (2SEG)

dcAlloc	dcCluster
dcFillZero	dcFlush
dcFree	dcGetPages
dcIsDirty	dcPgNumber
dcPxmDeclare	dcRead
dcSync	dcTrunc
dcWrite	lcCap
lcClose	lcFillZero
lcFlush	lcOpen
lcPushData	lcRead
lcSetRights	lcStat
lcTrunc	lcWrite
MpCreate	MpGetAccess
MpPullIn	MpPushOut
MpRelease	pageIoDone
pageMap	pagePhysAddr
pageSetDirty	pageSgId
pageUnmap	PxmClose
PxmGetAcc	PxmOpen
PxmPullIn	PxmPushOutAsyn
PxmRelAccLock	PxmStat
PxmSwapOut	rgnFlush
rgnInit	rgnInitFromDtCache
rgnMap	rgnMapFromDtCache
sgFlush	sgRead
sgStat	sgSyncAll
sgWrite	vmFlush

Device Driver Interfaces (9DDI)

ata	bench	bus
busFi	buscom	busmux
diag	disk	diskStat
ether	etherStat	flash
flashCtl	flashStat	generic_ata
gpio	hsc	isa
keyboard	mii	mngt
mouse	netFrame	nvram
pci	pciFi	pcimngr
pciswap	pcmcia	phy
quicc	ric	rtc
timer	uart	uartStat
vme	wdtimer	

Driver to Kernel Interfaces (9DKI)

	dataCacheBlockFlush
dataCacheBlockFlush_powerpc	dataCacheBlockInvalidate
dataCacheBlockInvalidate_powerpc	dataCacheFlush
dataCacheInvalidate	dataCacheInvalidate_powerpc
dcacheBlockFlush	dcacheBlockFlush_usparc
dcacheFlush	dcacheFlush_usparc
dcacheLineFlush	dcacheLineFlush_usparc
DISABLE_PREEMPT	dtreeNodeAlloc
dtreeNodeChild	dtreeNodeDetach
dtreeNodeFind	dtreeNodeFree
dtreeNodeParent	dtreeNodePeer

dtreeNodeRoot	dtreePropAdd
dtreePropAlloc	dtreePropAttach
dtreePropDetach	dtreePropFind
dtreePropFindNext	dtreePropFree
dtreePropLength	dtreePropName
dtreePropValue	eieio
eieio_powerpc	ENABLE_PREEMPT
hrTimerFrequency	hrTimerFrequency_powerpc
hrTimerFrequency_usparc	hrTimerFrequency_x86
hrTimerPeriod	hrTimerPeriod_powerpc
hrTimerPeriod_usparc	hrTimerPeriod_x86
hrTimerValue	hrTimerValue_powerpc
hrTimerValue_usparc	hrTimerValue_x86
hrt	hrt_powerpc
hrt_usparc	hrt_x86
icacheBlockInval	icacheBlockInval_usparc
icacheInval	icacheInval_usparc
icacheLineInval	icacheLineInval_usparc
imsIntrMask_f	imsIntrUnmask_f
instCacheBlockInvalidate	instCacheBlockInvalidate_powerpc
instCacheInvalidate	instCacheInvalidate_powerpc
ioLoad16	ioLoad16_x86
ioLoad32	ioLoad32_x86
ioLoad8	ioLoad8_x86
ioRead16	ioRead16_x86
ioRead32	ioRead32_x86
ioRead8	ioRead8_x86
ioStore16	ioStore16_x86
ioStore32	ioStore32_x86
ioStore8	ioStore8_x86

ioWrite16	ioWrite16_x86
ioWrite32	ioWrite32_x86
ioWrite8	ioWrite8_x86
load_sync_16_usparc	load_sync_32_usparc
load_sync_64_usparc	load_sync_8_usparc
loadSwap_16	loadSwap_32
loadSwap_64	loadSwap_sync_16_usparc
loadSwap_sync_32_usparc	loadSwap_sync_64_usparc
loadSwapEieio_16	loadSwapEieio_16_powerpc
loadSwapEieio_32	loadSwapEieio_32_powerpc
store_sync_16_usparc	store_sync_32_usparc
store_sync_64_usparc	store_sync_8_usparc
storeSwap_16	storeSwap_32
storeSwap_64	storeSwap_sync_16_usparc
storeSwap_sync_32_usparc	storeSwap_sync_64_usparc
storeSwapEieio_16	storeSwapEieio_16_powerpc
storeSwapEieio_32	storeSwapEieio_32_powerpc
svAsyncExcepAttach	svAsyncExcepAttach_usparc
svAsyncExcepDetach	svAsyncExcepDetach_usparc
svDeviceAlloc	svDeviceEntry
svDeviceEvent	svDeviceFree
svDeviceLookup	svDeviceNewCancel
svDeviceNewNotify	svDeviceRegister
svDeviceRelease	svDeviceUnregister
svDkiClose	svDkiEvent
svInitLevel	svIoRemap
svDkiOpen	svDkiThreadCall
svDkiThreadCancel	svDkiThreadTrigger
svDriverCap	svDriverEntry
svDriverLookupFirst	svDriverLookupNext

svDriverRegister	svDriverRelease
svDriverUnregister	svIntrAttach
svIntrAttach_powerpc	svIntrAttach_usparc
svIntrAttach_x86	svIntrCtxGet
svIntrCtxGet_powerpc	svIntrCtxGet_usparc
svIntrCtxGet_x86	svIntrDetach
svIntrDetach_powerpc	svIntrDetach_usparc
svIntrDetach_x86	svMemAlloc
svMemFree	svPhysAlloc
svPhysFree	svPhysMap
svPhysMap_powerpc	svPhysMap_usparc
svPhysMap_x86	svPhysUnmap
svPhysUnmap_powerpc	svPhysUnmap_usparc
svPhysUnmap_x86	svSoftIntrAttach_usparc
svSoftIntrDetach_usparc	svTimeoutCancel
svTimeoutGetRes	svTimeoutSet
svTimerIntrAttach_usparc	svTimerIntrDetach_usparc
swap_16	swap_32
swap_64	swapEieio_16
swapEieio_16_powerpc	swapEieio_32
swapEieio_32_powerpc	usecBusyWait
vmMapToPhys	vmMapToPhys_powerpc
vmMapToPhys_usparc	vmMapToPhys_x86
vmMapToPhys_x86	

Driver Implementations (9DRV)

amd29xxx	atadisk	benchns16550
bench_softint	bench_tbDec	buscom_loopback

buscom_nexus	buseth	busmux
cheerio	dec2115x	dec2155x
dec21x4x	ebus	e100
el3	epfpld	epic100
falcon	falcon_flashCtl	fccEther
flashdisk	fpga	generic_ata
gpiohsc	harrier	hawk
i8042	i8254	i8259
intel28fxxx	isabios	isaFi
isapci	m48txx	mc146818
muxtst	ne2000	ns16550
openpic	pcibios	pcicom_dec2155x_master
pcicom_dec2155x_slave	pcicom_host	pciconf
pcienum	pciFi	pcimngr
pciswap	pitTimer	quicc8260
quicc8xx	raven	raven_wdtimer
ric	rio	sabre
sccEther	sccUart	simba
siuWdt	smc1660	smc91xx
smcUart	sym53c8xx	tbDec
universe	univRemCom	usparchsc
vt82c586	vt82c586_ata	w83c553
w83c553_ata	z8536	z85x30

Standard C Library Functions (3STDC)

These services are available to applications that do not link with the POSIX subsystem.

Note – When you are not using the POSIX subsystem, only basic C++ programming is permitted. In particular, the STDC++ library is not available outside the POSIX subsytem, restricting applications from the use of `io` and `exception` classes.

---

abort	abs	atexit
atof	atoi	atol
bcmp	bcopy	bsearch
bzero	calloc	div
exit	fabs	ffs
free	getchar	getenv
getopt	getsubopt	getw
htonl	htons	isinf
isnan	isprint	labs
ldexp	ldiv	longjmp
malloc	memccpy	memchr
memcmp	memcpy	memmove
memory	memset	mkstemp
mktemp	mktime	modf
ntohl	ntohs	perror
printerr	printf	putc
putchar	putenv	puts
putw	qsort	rand
rand_r	random	realloc
realpath	regcomp	regerror
regex	regexec	regfree
remove	rewind	rindex
scandir	scanf	setbuf
setenv	setjmp	setstate
setvbuf	snprintf	sprintf
srand	srandom	sscanf

stdarg	strcasecmp	strcat
strchr	strcmp	strcoll
strcpy	strcspn	strdup
strerror	strftime	string
strlen	strncasecmp	strncat
strncmp	strncpy	strpbrk
strrchr	strsep	strspn
strstr	strtod	strtok
strtok_r	strtol	strtoul
strxfrm	swab	sys_errlist
sys_nerr	taddr2uaddr	tempnam
thread_once	time	tmpfile
tmpnam	toascii	tolower
toupper	tzset	uaddr2taddr
ungetc	unlocked	unsetenv
vfprintf	vprintf	vsnprintf
vsprintf		

General Microkernel APIs

- IPC
- MIPC
- LAP
- Black Box

Glossary

absolute binary	A binary file where all addresses have been resolved and computed.
actor	See ChorusOS actor.
ADMIN	The ADMIN actor administers high-level operating system services. See also C_OS.
application	A generic term to describe code running on ChorusOS (either a process or an actor).
asynchronous communication mode	The sender of an asynchronous message is only blocked for the time taken for the system to process the message locally. There is no guarantee that the message has been deposited at the destination. See also synchronous communication mode.
basic profile	A lightweight configuration of the ChorusOS operating system. This configuration is pre-built and part of the binary delivery. The configurator utility can be used to change the profile. See also configurator, and extended profile.
binary delivery	The binary delivery of Sun Embedded Workshop includes the development environment for ChorusOS and the ChorusOS operating system in binary, along with the BSPs for the reference target boards in the supported target family. See also source delivery.
black box	Black box provides a means for tracing the activity of the system and applications using multiple in-memory buffers managed by the system.
board	See target board.

Board Support Package	The BSP (Board Support Package) is the set of files that can be customized to run ChorusOS on specific board architectures. It is provided in source with the binary product, allowing you to port to another board within the same target family. The BSP contains the boot and the generic and processor specific drivers required for your board.
boot	The boot provides system boot, system reboot, and debug services.
boot kernel interface (BKI)	The BKI is the interface between the microkernel and the boot. It is a set of rules used when the microkernel is launched. It is split into a generic part that is common to all target families and a specific part that applies to a single target family.
bootMonitor	See initial loader.
BSP	See Board Support Package.
c_actor	No longer used. See POSIX process.
C_INIT	The command interpreter of the ChorusOS operating system invoked by the system when it is loaded. `C_INIT` can be accessed by `RSH` or `LOCAL_CONSOLE`.
C_OS	An actor that manages input/output and provides the POSIX interface to applications. C_OS also manages file system, networking, and shared memory.
CBDI	See Common Bus Driver Interface (CBDI).
CDC	Connected Device Configuration, is part of J2ME. See J2ME.
ChorusOS actor	A unit of execution for an application. Serves as the encapsulation unit to associate all system resources used by the application and the threads running within the actor. ChorusOS actors are restricted to the microkernel API, that is, the APIs exported by the microkernel to run either: ■ The "system" software, that is, hardware-related software (BSPs), or software related to OS services (networking, file administration, and so on). ■ Services provided by the ChorusOS microkernel that cannot be used by POSIX processes. Exceptions are IPC, MIPC, and LAP, which can be used by both ChorusOS actors and POSIX processes. ChorusOS actors can be *user* and *supervisor*, or *trusted* and *non-trusted*. See also POSIX process, ChorusOS 4.x legacy application, user actor, supervisor actor.
ChorusOS 4.x legacy application	Applications developed for 4.x releases of the ChorusOS operating system. ChorusOS 5.0 fully supports 4.x legacy applications to help 4.x

users transition to 5.x, but users are not encouraged to develop new 4.x legacy applications since no compatibility is planned beyond ChorusOS 5.0.

See also POSIX process, and ChorusOS actor.

Common Bus Driver Interface (CBDI) An interface that is independent of bus type, offering a set of services common for all bus classes.

component The ChorusOS operating system is made up of several components that can be specified when building the system image. Some components are mandatory for the system, such as the kernel, DRV, and the bsp components. Other components include, os, tools, the X11 libraries, and the examples.

configurator A command-line tool used to configure the ChorusOS system image.

See also Ews, and configuring a system image.

configure A tool to configure your build directory with the paths of the components to be included in your system image.

configuring a system image This refers to including optional components in your system image, turning features on or off, and changing the value of tunable parameters and environment variables in your ChorusOS system image. This can be carried out using different tools, including configure, configurator, Ews. The configuration is stored in a number of different ECML files.

core dump Core dump allows offline, post-mortem, analysis of actors or processes that are killed by exceptions.

core executive The core executive provides essential services to support real-time applications. It supports multiple, multi-applications running in both *user* and *supervis*or memory space.

It implements the basic ChorusOS execution model and provides the framework for all other features that can be configured. Every system includes a core executive.

The core executive exports the basic set of microkernel abstractions and services:

- The unit of application modularization (actor)
- The unit of execution (thread)
- Thread control operations
- Exception management services
- A minimal interrupt management service

No synchronization, scheduling, time, communication, or memory management policies are provided by the core executive. These

	policies and services are provided by additional features, which the user may select depending on particular hardware and software requirements.
CVM	CVM (C Virtual Machine), is a compact Java virtual machine, part of the J2ME provided with ChorusOS. It allows you to implement Java applications on a ChorusOS system. See also J2ME.
device driver framework	A framework that is provided with ChorusOS that allows programmers to develop device drivers on top of a binary distribution of the ChorusOS operating system. The device driver framework provides a structured and easy-to-use environment to develop both new drivers and client applications for existing drivers.
device driver interface (DDI)	The device driver interface defines the interfaces between different layers of device drivers in the driver hierarchy. Typically, an API is defined for each class of device or bus, as a part of the DDI.
device kernel interface (DKI)	The device kernel interface defines all services provided by the microkernel to driver components. Following a layered interface model, all services implemented by the DKI are called by the drivers, and take place in the microkernel.
device tree	The ChorusOS device tree is a data structure that describes hardware topology and device properties. It is constructed in terms of parent/child relationships between devices. Device properties are specific to each device and defined in name/value pairs. The initial device tree is built by the bootstrap program using the DKI tree browsing API.
DHCP	See Dynamic Host Configuration Protocol.
DNS	See Domain Name Service (DNS).
Domain Name Service (DNS)	Standard naming architecture used for naming on the Internet Protocol (IP).
DRV	DRV is the ChorusOS component that contains all generic microkernel drivers and all board specific drivers.
dynamic loading	On actor start-up, the runtime linker loads the actor or process and performs the necessary relocations. It also loads and links the actor dependencies (the required dynamic libraries).
dynamic and static identifiers	See static and dynamic identifiers.
Dynamic Host Configuration Protocol	A communications protocol that lets network administrators manage centrally and automate the assignment of Internet Protocol (IP) addresses in an organization’s network.

embedded actor	An actor can be embedded, which means that it becomes part of the system image and is launched at boot time. Embedded actors that are not boot actors can be loaded by other embedded actors using the `afexec()` system call. Typically, drivers are embedded actors. Drivers do not require the services of the POSIX subsystem and can be started dynamically through the `afexec()` system call.
Embedded Component Mark Language (ECML)	The Embedded Component Mark Language provides a unique interface to the configurable information of the ChorusOS operating system. ECML is defined using XML syntax, making it benefit from existing XML tools, especially browsers and parsers. The configuration of ChorusOS is defined in a number of EMCL files that can be edited using the configurator and Ews tools.
Ews	A graphical configuration tool used to view and modify the configuration of a ChorusOS operating system through a set of ECML configuration files. See also configuring a system image, Embedded Component Mark Language (ECML), configurator.
execution actor	The actor on behalf of which the thread currently executes. Any thread has the right to access and manage the resources of its execution actor. See also home actor, and ChorusOS actor.
extended profile	A full configuration of the ChorusOS operating system, selected at installation. See also basic profile.
feature	Services that can be selected when you configure your system image. You can configure features using either the command-line utility, configurator, or the graphical utility, Ews. Some features have tunable parameters associated with them. See also configurator, Ews, and tunable parameter.
flat memory	Simple memory allocation service. The microkernel and all applications run in one unique, unprotected address space. Virtual addresses match physical addresses directly. This is used for systems without an MMU or when the MMU is bypassed.
home actor	The actor in which the thread was created. The home actor of a thread is constant over the life of the thread. Also called owning actor.
host	A machine running the Solaris operating environment that hosts the development environment, Sun Embedded Workshop, for building and configuring a ChorusOS system image and applications that run on it.

	See also Sun Embedded Workshop software (SEW).
hot restart	Mechanism for restarting applications or entire systems if a serious error or failure occurs. This feature addresses the high-availability requirements of the operating system, reducing the time taken for a failed system or component to return to service. See also Hot Restart Controller (HR_CTRL).
Hot Restart Controller (HR_CTRL)	An actor that monitors restartable actors to detect abnormal termination and automatically take the appropriate restart action. Abnormal termination includes unrecoverable errors, such as, division by zero, a segmentation fault, unresolved page fault, or invalid operation code. See also hot restart, restartable actors, and persistent memory.
hot swap	A ChorusOS feature that allows you to remove and replace a board from an instance of the ChorusOS operating system without having to shut down the system. See also Hot Swap Controller (HSC).
Hot Swap Controller (HSC)	A board-dependent layer that handles the `ENUM#` signal, which notifies the system of an insertion or removal event specified by cPCI hot swap. See also hot swap, and PCI.
HR_CTRL	See Hot Restart Controller (HR_CTRL).
ICMPv6	See Internet Control Message Protocol.
Industry Standard Architecture (ISA)	Industry Standard Architecture is a standard bus (computer interconnection) architecture that is associated with the IBM AT motherboard.
initial loader	The initial loader enables a system image to be located on the network or local device, booted on the target board. The initial loader provided by ChorusOS is `bootMonitor`. This functionality is also provided by the target firmware (depending on your target).
Internet Control Message Protocol	ICMP is a message control and error-reporting protocol between a host server and a gateway to the Internet. ICMP uses Internet Protocol (IP) datagrams, but the messages are processed by the IP software and are not directly apparent to the application user.
inter-process communication (IPC)	IPC provides synchronous (RPC) and asynchronous communication features allowing threads to communicate and synchronize when they do not share memory. Communications rely on the exchange of messages through ports.

	See also Remote Procedure Call (RPC), asynchronous communication mode, static and dynamic identifiers, and MIPC.
IPC	See inter-process communication (IPC).
IPv4 and IPv6	Internet Protocol versions 4 and 6.
ISA	See Industry Standard Architecture (ISA).
J2ME	Java 2 Platform Micro Edition technology covers the range of embedded devices. The ChorusOS port of this technology includes the Foundation profile and the Connected Device Configuration (CDC), which provides a virtual machine and basic class libraries to support Java language applications on consumer electronic and embedded devices.
JNI	Java Native Interface
JPDA	Java Platform Debugger Architecture
JVMDI	Java Virtual Machine Debug Interface
JDWP	Java Debug Wire Protocol
LAP	See Local Access Point (LAP).
Lightweight Directory Access Protocol (LDAP)	LDAP in ChorusOS provides access to X.500 directory services. These services can be a standalone part of a distribution service. An LDAP client library is available with ChorusOS that provides programmatic access to the LDAP.
Local Access Point (LAP)	A generic software interface for microkernel-mediated calls from one actor to another on the local site. A LAP is a microkernel object that represents a handler function that has been exported by a supervisor actor for invocation by client actors. It is a super-fast inter-actor communication facility provided by the operating system.
makefile target	An optional argument supplied to the `make` command that corresponds to a set of commands in the makefile. This enables you to specify more than one set of actions in a single makefile. For example, the `make kernonly` command allows you to build a system image containing just the microkernel and some OS services.
Management Information Base (MIB)	The Management Information Base is the language to describe the parameters managed through the SNMP protocol.
message handlers	Message handlers are an alternative to threads. Instead of creating threads explicitly, an actor can attach a handler (a routine in its address space) to the port. When a message is delivered to the port, the handler is executed in the context of a thread provided by the microkernel.

microkernel	The microkernel is the core of the ChorusOS operating system. It provides a minimum set of interfaces that are used by the remainder of the operating system. You must always include the microkernel in your system image.
mini profile	A profile containing the absolute minimum features for a `bootMonitor` image. See also basic profile, and extended profile.
MIPC	The MIPC provides a fast communication facility based on exchange of messages through shared mailboxes. See also inter-process communication (IPC).
monitor	A monitor is a set of instructions in which only one thread may execute at a time.
MSDOSFS	MS-DOS File System.
mutex	Sleep locks provided in the form of mutual exclusion locks. These are data structures allocated in the client actors' address spaces.
netboot	Application management utility used to boot a ChorusOS system image using TFTP when the target does not provide an embedded booting facility.
Network Information Service (NIS)	A network naming and administration system for small networks. Using NIS, each host client or server computer in the system has knowledge about the entire system. A user at any host can get access to files or applications on any host in the network with a single user identification and password. NIS is similar to the Internet's domain name system (DNS) but somewhat simpler and designed for a smaller network.
NTP	Network Time Protocol
operating system	The ChorusOS operating system provides a set of interfaces that enables you to run applications on various hardware configurations. The ChorusOS operating system contains the microkernel, board support package and higher level services, including I/O and file system support.
owning actor	See home actor.
patting	The process of resetting a watchdog timer. This process is usually managed by a dedicated user-level process. If not, an interrupt handler provided by the system is invoked. See also watchdog timer.
PCI	Peripheral Component Interconnect (personal computer bus)

PD	The PD (Private Data manager) implements the per thread data interface between the microkernel subsystems, such as the the C_OS subsystem.
persistent memory	The hot restart mechanism relies on persistent memory to store data that can persist across an actor or site restart. It is used internally by the system to store on the RAM the actor image (text and data) from which a hot restartable actor can be reconstructed. See also Persistent Memory Manager (PMM).
Persistent Memory Manager (PMM)	The PMM (Persistent Memory Manager) implements the persistent memory interface. It is a ChorusOS actor that manages the allocation and freeing of persistent memory blocks. The PMM is automatically included in the system image when the `HOT_RESTART` feature is activated. See also persistent memory.
platform	See target board.
Point-to-Point Protocol (PPP)	A protocol for communication between two computers using a serial interface, typically a personal computer connected by phone line to a server. Essentially, it packages your computer's TCP/IP packets and forwards them to the server where they can actually be put on the Internet.
Portable Operating System Interface (POSIX)	A standard set of APIs for portable multithreaded programming.
ports	An address to which messages can be sent and that has a queue holding the messages received by the port but not yet consumed by the threads. Ports are attached to actors. Ports can be assembled into groups adding a multicast facility. See also message handlers.
POSIX process	ChorusOS 5.0 provides a complete set of POSIX APIs and allows you to create POSIX-compatible applications. These applications are called POSIX processes. A process is the unit of encapsulation of the POSIX subsystem. Processes can be either *user* or *supervisor*. See also ChorusOS actor, and ChorusOS 4.x legacy application.
PPP	See Point-to-Point Protocol (PPP).
process	See POSIX process.
process dump	`pdump` is a `C_INIT` built-in command which dumps a core image of one or more processes as an ELF format file. See also C_INIT.

profile	The ChorusOS operating system provides profiles that are used to set up an initial configuration. The profiles are called basic, extended, and mini. These profiles include or remove certain features in the system. You can use one of these configuration profiles as the initial configuration for your system, and add or remove specific feature options using the configurator utility. See also basic profile, extended profile, mini profile, and configurator.
protected memory	Memory allocation service supporting multiple address spaces and region sharing between different address spaces. Targeted at critical and non-critical real-time applications where memory protection is mandatory. This is used for systems with MMU, address translation, and where applications benefit from the flexibility and protection of separate address spaces.
protection identifier	The IPC allocates a protection identifier to each actor and to each port. The structure of the protection identifier is fixed, but IPC does not associate any semantics to their values.
relocatable actor	On actor start-up, the runtime linker loads the actor and performs the necessary relocations. In ChorusOS 5.0, all dynamic actors are relocatable.
relocatable binary	A binary file that can be loaded or relocated at any address.
Remote Procedure Call (RPC)	This protocol allows the construction of client-server applications using a demand/response protocol with management of transactions. The client is blocked until a response is returned from the server, or a user-defined optional timeout occurs. RPC guarantees at-most-once semantics for the delivery of the request. See also inter-process communication (IPC), and asynchronous communication mode.
restartable actors	Any actor that can be restarted rapidly without accessing stable storage, when it terminates abnormally. A restartable actor is restarted from an actor image that comprises the actor's text and initialized data regions, stored in persistent memory. See also restart group, hot restart, hot restart controller, and persistent memory.
restart groups	A group of cooperating restartable actors that can be restarted in the event of the failure or abnormal termination of one or more actors within the group. When one actor in the group fails, all actors in the group are stopped and then restarted either directly by the system or indirectly by spawning. These groups are usually mutually exclusive. A restart group is created dynamically in ChorusOS when a direct actor is declared to be a member of the group. Therefore, each group contains at least one direct actor. An indirect actor is always a member of the same group as the actor that spawned it. This applies only when the `HOT_RESTART` feature is applied.

RPC	See Remote Procedure Call (RPC).
scheduler	A scheduler provides scheduling policies, which are rules, procedures, or criteria used in making process scheduling decisions. The scheduling policies available in the ChorusOS operating system include FIFO (first-in-first-out) and RR (round robin).
semaphore	An integer counter and an associated thread-wait queue. When initialized, the semaphore counter receives a user-defined positive or null value.
SEW	See Sun Embedded Workshop software (SEW).
software interrupts	Interrupts supported by the DKI and DDI. These are not initiated by a hardware device, but rather by the software. Handlers for these must be added to and removed from the system. Software interrupt handlers run in the interrupt context and therefore can be used to do many of the tasks that belong to the interrupt handler.
source delivery	The source delivery of the Sun Embedded Workshop includes the ChorusOS operating system in source files. Note that the development tools are delivered in binary. See also binary delivery.
static and dynamic identifiers	These are communication entities that are part of the uniform global naming scheme of inter-process communication (IPC). Static identifiers are provided to the system by the application. Dynamic identifiers are returned by the system to the application. See also inter-process communication (IPC).
Sun Embedded Workshop software (SEW)	The development environment for the ChorusOS operating system. This software includes the tools necessary to configure and build a ChorusOS operating system and the applications to run on it. See also host.
supervisor actor	This is a type of actor that shares a common but partitioned address space with other supervisor actors. These actors can execute privileged hardware instructions depending on the underlying hardware. See also ChorusOS actor, and user actor
supervisor process	A POSIX process that shares a common supervisor address space with other processes, which themselves are necessarily supervisor processes. See also user process.
synchronous communication mode	The sender of a synchronous message is blocked until its request message has been received, answered, replied to, and the reply has been received. The whole loop is guaranteed to perform and errors or

	failures are detected and reported. The synchronous communication mode is sometimes called RPC (Remote Procedure Call). See also Remote Procedure Call (RPC), and asynchronous communication mode
sysadm.ini file	A script file embedded in the file system boot image. This file is executed as the last step of the system initialization and is used by `C_INIT` after the system is initialized. You can customize it to run selected applications directly on system start-up.
system dump	Enables the system to collect data in case of a system failure. Data collection is defined as the content of the black box buffers. On a system crash, this data is copied to a persistent memory area, or dump area, based on the hot restart feature of the ChorusOS operating system. See also black box.
system image	A system image is an instance of the ChorusOS operating system. It is made up of binary or executable files that define the operating system and initial application processes. After you have built and booted your system image, you are ready to start using the ChorusOS operating system.
target	See target board.
target board	The physical hardware board on which a ChorusOS system will run. The board must be in one of the supported target families to be able to run a ChorusOS system. See also target family.
target family	A set of target boards with processors that are electronically different but are identical in their interactions with the operating system. The target families supported by the ChorusOS operating system are as follows: ■ UltraSPARC IIi/IIe ■ Intel x86/Pentium ■ Motorola PowerPC 750/765/74x0 processors and PowerQUICC II (mpc8260) microcontrollers ■ Motorola PowerQUICC I (mpc8xx) microcontrollers When porting a ChorusOS system, you will be porting it to a new board in a supported target family. See also target board.
TCK	Java Technology Conformance Kit

timer (general interval timer)	High-level timer service for both user and supervisor actors. This feature uses the concept of a timer object in the actor environment. Timers are created and deleted dynamically. All high-level timer operations such as setting, modifying, querying, or canceling pending timeouts, refer to timer objects.
tunable parameter	Tunables are parameters that you configure in your ChorusOS system image by setting an integer value. For example, timeout delays or maximum file sizes. Some tunable parameters are linked to a feature and are therefore only configurable if the feature is on. These are called static tunable parameters. In addition, there are system environment variables or dynamic tunable parameters that allow you to set the environment, a set of `VARIABLE=value` character strings defining the values for environment variables (for example, the IP address) read by the applications. See also feature.
user actor	This is a type of actor that has separate and protected address spaces. See also ChorusOS actor, and supervisor actor
user process	A POSIX process that runs in its own private protected address space. See also supervisor process.
virtual memory	Memory allocation service supporting multiple, protected address spaces. On systems with secondary storage, applications can use much more virtual memory than the memory physically available. This module is specifically designed to implement distributed UNIX subsystems on top of the microkernel.
virtual timer	A virtual timer is responsible for all functions pertaining to measurement and timing of thread execution. It exports a number of functions used typically by higher-level operating systems such as UNIX.
watchdog timer	Mechanism provided with the ChorusOS operating system to detect hardware and operating system failures. Note that this feature relies on the hardware watchdog device available on modern boards. See also patting.

Index