



ChorusOS 4.0 Introduction

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

Part Number 806-0610-10
December 1999

Copyright 1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 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, ChorusOS, Sun Embedded WorkShop, 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.

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.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

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.

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, ChorusOS, Sun Embedded WorkShop, 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.

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 REPENDRE 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.



Contents

Preface 17

Part I Technical Overview

1. Technical Overview 25

Introduction to Sun Embedded Workshop 25

Sun Embedded Workshop Components 26

Supported Processor Families 26

Features and Benefits 26

Multi-platform Development Environment 27

Portable Binary System 27

Super-Configurability 27

High Availability 29

Support for Legacy APIs 30

Support for Java[™] Applications 30

Transparent Inter-Process Communication (IPC) 31

Operating System Components 31

The Core Executive 34

Optional Operating System Services 35

Configuring ChorusOS 51

The Extended Profile 51

	The Basic Profile	51
	Development Environment Components	51
	Debugging Architecture	53
	Management Utilities	53
	Development Lifecycle	54
	Installing Sun Embedded Workshop	54
	Developing an Application	55
	Developing a System	56
	Part II Using ChorusOS	
2.	Using ChorusOS	59
	The ChorusOS System Image	59
	Downloading the System Image	59
	Basic Environment	60
	Building an Application Actor	60
	Embedding your Actor in the System Image	61
	Running your Actor in the Basic Environment	62
	Extended Environment	62
	Communicating with the Target Using <code>rsh</code>	63
	Mounting the Host File System	63
	Security	64
	Running the “Hello World” Example	65
	Input/Output Management	66
	System Administration in the Extended Environment	67
	C_INIT Actor	67
	System Start-up	68
	Initialization Examples	68
3.	Configuring and Tuning	71
	Configuration Options	71

Feature Options	72
Configuration Profiles	72
Tunable Parameters	74
System Image Components	75
Configuration Files	75
Configuration Tools	76
Graphical Configuration Tool	76
Command-line Configuration Tool	86
Part III Programming Overview	
4. Programming Overview	95
ChorusOS Applications	96
Programming Conventions	96
General Principles	96
Application Programming Interfaces	98
Naming Conventions	98
Basic Environment APIs	98
Extended Environment API	99
Other APIs	100
Multithreading	101
Header Files	101
Developing ChorusOS Applications	103
make Environment	103
imake Environment	104
Examples	107
Using Dynamic Libraries	109
Static and Dynamic Linking	110
Building a Dynamic Library	111
Building a Dynamic Program	112

	Dynamic Programming	112
	Runtime Linker	113
	Examples	115
5.	Using Actors	121
	Actor Definition	121
	Naming Actors	123
	User and Supervisor Actors	123
	Loading Actors	126
	Boot Actors	126
	Loading Actors Dynamically	126
	Execution Environment of Actors	127
	Actor Context	127
	Standard Input/Output (I/O)	128
	Allocating Memory	130
	Terminating an Actor	130
	Spawning an Actor	130
6.	Multithreaded Programming with the ChorusOS Operating System	133
	Basic Multi-Thread Programming	133
	Thread Handling	135
	Getting a Thread Identifier	135
	Creating a Thread	135
	Deleting a Thread	139
	Synchronizing Threads	142
	Semaphores	143
	Mutexes	147
	Basic Scheduling Control	150
	Managing Per-Thread Data	154
	Threads and Libraries	158

7.	Memory Management	161
	Memory Region Descriptors	161
	Allocating and Freeing Memory Regions	163
	Sharing Memory Between Two Actors	167
8.	Inter-actor Communication	173
	Introduction	173
	Message Queues	174
	Local Access Points	184
	IPC	187
9.	Time Management	191
	Time Management Services	191
	Current Time	192
	Timers	193
	Part IV Debugging and Performance Profiling	
10.	System and Application Debugging	203
	Preparing the System for Symbolic Debugging	203
	Compiling for Debugging	203
	Enabling Debugging for Components Built with <code>imake</code>	204
	Enabling Debugging for Components Built with <code>mkmk</code>	206
	Configuring the Debug Agent	206
	Application Debugging Architecture	207
	Architecture Overview	208
	Setting up a Debugging Session	208
	RDBC Configuration and Usage	209
	System Debugging Architecture	210
	Architecture Overview	210
	Setting up a Debugging Session	211
	Starting and Configuring the ChorusOS DebugServer	212

	RDBS Configuration and Usage	217
	Concurrent System and Application Debugging	218
	Example XRAY/RDBS debug session	219
	Troubleshooting	222
	Sample XRAY Start-up Script	223
11.	Performance Profiling	225
	Introduction to Performance Profiling	225
	Preparing to Create a Performance Profile	227
	Configuring the System	227
	Compiling the Application	227
	Launching the Performance Profiled Application	227
	Running a Performance Profiling Session	228
	Starting the Performance Profiling Session	228
	Stopping the Performance Profiling Session	228
	Generating Performance Profiling Reports	229
	Analyzing Performance Profiling Reports	229
	Performance Profiler Description	231
	The Performance Profiling Library	232
	The Performance Profiler Server	232
	The Performance Profiling Clock	232
	Notes About Accuracy	232
A.	Configuring IPC	235
	Generic IPC Configuration	235
	IPC Feature Configuration	235
	Site Number Administration	236
	Specific IPC Configuration	237
	Remote IPC over Ethernet Data-link	237
	Remote IPC over VME Bus	239

Glossary 241

Index 247

Tables

TABLE P-1	Typographic Conventions	20
TABLE P-2	Shell Prompts	21
TABLE 1-1	Operating System Optional Components	32
TABLE 3-1	Feature settings in the <code>extended</code> and <code>basic</code> configuration profiles	72
TABLE 4-1	Compilation Options	103
TABLE 4-2	Imake build rules	104
TABLE 4-3	Imake packaging rules	106
TABLE 9-1	Time Management Service Availability	192
TABLE A-1	VME memory dedicated to IPC	239

Figures

Figure 1–1	Component-based Operating System Architecture	28
Figure 3–1	EWS User Interface	78
Figure 3–2	Kernel Configuration Displayed in HTML	86
Figure 5–1	User and Supervisor Address Spaces	124
Figure 6–1	A Multi-Threaded Actor	134
Figure 6–2	Two Threads Synchronizing with a Semaphore	144
Figure 7–1	Memory Region Allocation and Deallocation	163
Figure 7–2	Actors Sharing Memory	168
Figure 8–1	Creating a Message Space	176
Figure 8–2	Opening a Message Space	176
Figure 8–3	Allocating Messages from Pools	177
Figure 8–4	Posting Messages to Queues	178
Figure 8–5	Getting Messages from Queues	179
Figure 10–1	Application Debugging Architecture	208
Figure 10–2	System Debugging Architecture	211
Figure A–1	Device sub-tree representing the VME bus	240

Code Examples

CODE EXAMPLE 5-1	Getting Actor Privilege	125
CODE EXAMPLE 5-2	Using the C Library from an Actor	128
CODE EXAMPLE 5-3	Spawning an Actor	131
CODE EXAMPLE 6-1	Creating a Thread	136
CODE EXAMPLE 6-2	Deleting a Thread	140
CODE EXAMPLE 6-3	Synchronizing Using Semaphores	144
CODE EXAMPLE 6-4	Protecting Shared Data Using Mutexes	148
CODE EXAMPLE 6-5	Changing Scheduling Attributes	152
CODE EXAMPLE 6-6	Managing Per-Thread Data	155
CODE EXAMPLE 7-1	Allocating a Memory Region	164
CODE EXAMPLE 7-2	Sharing a Memory Region	169
CODE EXAMPLE 8-1	Communicating Using Message Spaces	179
CODE EXAMPLE 8-2	Creating and Invoking LAPs	184
CODE EXAMPLE 8-3	Communicating Using IPC	188
CODE EXAMPLE 9-1	Using Timers	195

Preface

This book introduces the features and components of Sun Embedded Workshop™ and the ChorusOS™ operating system. It explains how to use Sun Embedded Workshop and how to create an application that runs on the ChorusOS operating system.

Who Should Use This Book

Use this book if you are using Sun Embedded Workshop to develop an application that runs on a ChorusOS operating system. This book is also useful if you are evaluating Sun Embedded Workshop and the ChorusOS operating system.

Before You Read This Book

This book assumes that you have:

- A general understanding of embedded operating systems
- Knowledge of the C programming language (for Part III)

How This Book is Organized

Part I introduces the product and its components, and explains how the product can be used.

- Chapter 1 contains an overview of the product.

Part II explains how to use the ChorusOS operating system.

- Chapter 2 explains the use of the ChorusOS operating system.
- Chapter 3 explains how to configure and tune a ChorusOS operating system.

Part III describes how to develop an application that runs on the ChorusOS operating system.

- Chapter 4 is an overview of the tasks involved in developing an application.
- Chapter 5 explains how actors are used in an application.
- Chapter 6 explains how to use the multithreading services provided in the ChorusOS operating system.
- Chapter 7 explains how to use the memory management services provided in the ChorusOS operating system.
- Chapter 8 explains how to use the inter-actor communication services provided in the ChorusOS operating system.
- Chapter 9 explains how to use the time management services provided in the ChorusOS operating system.

Part IV describes debugging and performance profiling on the ChorusOS operating system.

- Chapter 10 explains how to debug the ChorusOS operating system and applications.
- Chapter 11 explains how to analyze the performance of the ChorusOS operating system and its applications by generating a performance profile report.

The *Glossary* is a list of words and phrases found in this book and their definitions.

Related Books

ChorusOS 4.0 Installation Guide explains how to download and install Sun Embedded Workshop. *ChorusOS Release Notes* contains information about new features and restrictions in this release of the product.

See the appropriate document in the *ChorusOS 4.0 Target Family Documentation Collection* for instructions explaining how to build and run the ChorusOS operating system on supported hardware.

The following books describe how to use Sun Embedded Workshop components:

- *ChorusOS 4.0 File System Administration Guide* explains how to use the file systems provided with the ChorusOS operating system. It includes information about using the NFS server.
- *ChorusOS 4.0 Network Administration Guide* explains how to use the networking capabilities of the ChorusOS operating system.

The Mentor Graphics Corporation *XRAY Debugger for ChorusOS* includes documentation explaining how to debug a ChorusOS application. XRAY is the reference debugger for use with the ChorusOS operating system.

The following books contain information about advanced programming with Sun Embedded Workshop:

- *ChorusOS 4.0 Porting Guide* explains how to port the ChorusOS operating system to another target board.
- *ChorusOS 4.0 Device Driver Framework Guide* describes the device driver architecture of the ChorusOS operating system and explains how to add a new driver.
- *The ChorusOS 4.0 Hot Restart Programmer's Guide* describes how to develop applications to use the hot restart functionality of the ChorusOS operating system.
- *ChorusOS 4.0 Flash Guide* describes the support for flash memory provided in the ChorusOS operating system and explains how to use it.
- *ChorusOS 4.0 Production Guide* describes the organization of the source code and explains how to use it.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks selected 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.comSM 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 <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name%</code> you have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine_name%</code> su Password:

TABLE P-1 Typographic Conventions *(continued)*

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

Shell Prompts

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	<i>machine_name%</i>
C shell superuser prompt	<i>machine_name#</i>
Bourne shell and Korn shell prompt	<i>\$</i>
Bourne shell and Korn shell superuser prompt	<i>#</i>

PART I

Technical Overview

Technical Overview

This chapter contains an overview of Sun Embedded Workshop™.

- “Introduction to Sun Embedded Workshop” on page 25 provides a high-level overview of Sun Embedded Workshop.
- “Features and Benefits” on page 26 describes the key features of the product and why they are useful to you.
- “Operating System Components” on page 31 provides an overview of the operating system and its configurable components.
- “Development Environment Components” on page 51 contains a summary of the tools provided to help you develop an application or system using Sun Embedded Workshop.

Introduction to Sun Embedded Workshop

Sun Embedded Workshop provides the ChorusOS™ operating system and a complete development environment for creating an application that runs on the ChorusOS operating system or an embedded system based on the ChorusOS operating system.

High-performance and high-availability, combined with a simple, flexible configuration mechanism make Sun Embedded Workshop particularly well-adapted for developing and deploying a wide range of telecommunications, data communications, and consumer applications.

Sun Embedded Workshop Components

Sun Embedded Workshop contains the following components:

- The ChorusOS operating system. See “Operating System Components” on page 31 for information about the operating system components.
- A complete development environment for creating applications or systems that use this operating system. See “Development Environment Components” on page 51 for information about the development environment components.

Supported Processor Families

This release of Sun Embedded Workshop is available for the following development platforms:

- Solaris operating environments, supporting the following targets:
 - x86, Pentium
 - Motorola PowerPC 60x and 750 processor family (ppc60x)
 - Motorola PowerQUICC I (mpc8xx) and PowerQUICC II (mpc8260) microcontrollers
 - UltraSPARC III
- Windows NT platforms, supporting the following targets:
 - Motorola PowerPC 60x and 750 processor family (ppc60x)

See the *ChorusOS Release Notes* for the latest information about supported target platforms.

Features and Benefits

This section contains a summary of the key features and benefits of Sun Embedded Workshop.

Multi-platform Development Environment

Sun Embedded Workshop provides complete support, tools, and libraries for developing C and C++ applications on a range of supported platforms. Development takes place on one system (the host), even though the software will eventually run on a very different device (the target), or on a variety of targets.

Sun Embedded Workshop 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.

Portable Binary System

For each supported processor family, Sun Embedded Workshop 4.0 comes with the implementation of at least one target platform and provides a complete set of well defined interfaces allowing you to port the ChorusOS operating system to other target boards. The Boot Kernel Interface (BKI) and Device Driver Interface (DDI) available in the binary release of ChorusOS allows you to customize the boot method and to add new drivers.

Super-Configurability

Sun Embedded Workshop 4.0 uses a flexible, component-based architecture that allows different services to be configured into the runtime instance of the ChorusOS operating system.

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 very accurately configured to meet the exact needs of a given application or environment, saving on memory and improving performance.

The core executive can support multiple, independent applications running in both user and supervisor memory space.

This flexible architecture is shown in Figure 1–1. Detailed descriptions of the optional features for the ChorusOS operating system are provided in “Operating System Components” on page 31.

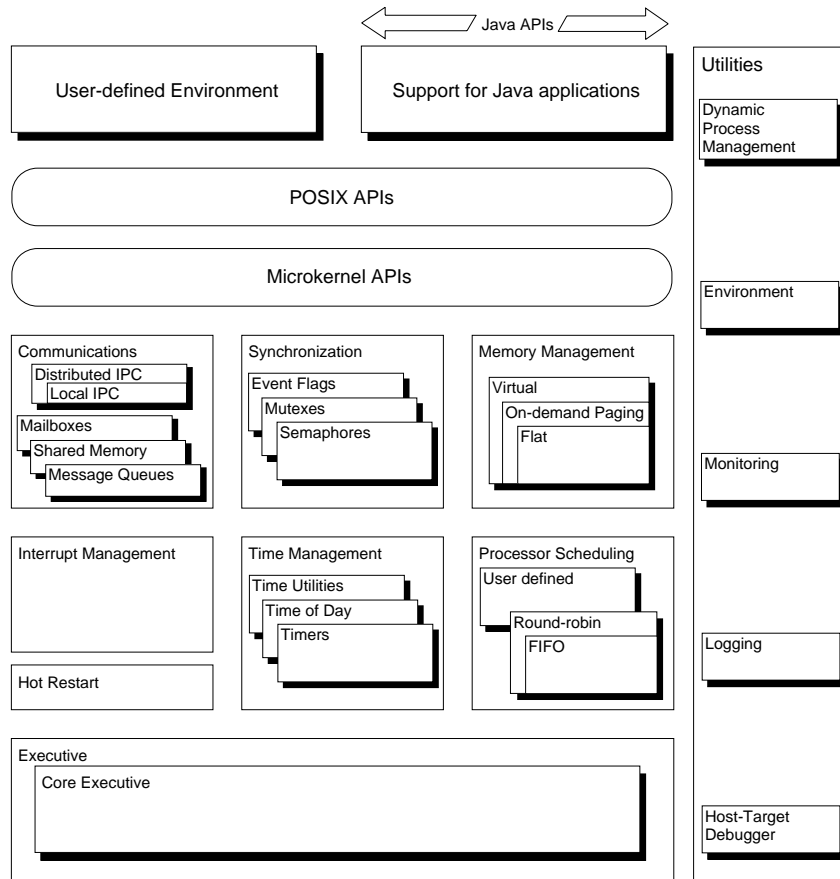


Figure 1-1 Component-based Operating System Architecture

By taking advantage of the component-based architecture, the application designer 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.

High Availability

Building large, highly-available systems is a complex and challenging undertaking that has required significant advances in design, implementation, and testing methodologies. For example, the telecommunications industry faces severe reliability and availability constraints imposed by international standards and market pressure. Yet, until recently, very few commercially available operating systems could provide the appropriate level of support to be able to offer true 7 by 24 operation.

The ChorusOS operating system incorporates several features that successfully address the needs of this demanding market, including:

- Memory Protection
- Hot Restart
- Dynamic Reconfiguration

Memory Protection

Different applications can run in different memory address spaces 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.

Hot Restart

An important benefit of the ChorusOS operating system is its hot restart capability, which provides one of the fastest mechanisms available in the industry today 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 from 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 the conventional cold restart technique and retains the critical information that allows an application to be reconstructed quickly with little or no interruption of 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.

For detailed information about the hot restart feature, refer to the *ChorusOS 4.0 Hot Restart Programmer's Guide*.

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 inter-process communication (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), new features such as call forwarding (or follow me) can be added without interrupting the basic telephone service and without reconfiguring the entire telephone network.

Support for Legacy APIs

One of the challenges facing software developers working in the telecommunications and data communications industries is the need to select the most appropriate of the proliferation of public standards and proprietary solutions available to them. By investing in a given solution, application vendors and service providers can quickly find themselves locked into a legacy API that once seemed to offer state-of-the-art functionality. In some cases, finding that they can no longer keep pace with emerging technology, they are forced to either fall behind or to abandon their original investment.

The ChorusOS operating system offers software developers a way to protect their existing investments, while providing a smooth migration path to new platforms running the ChorusOS operating system. It does this by:

- Providing a way for applications to handle traps, which allows software developers to create proprietary subsystems to emulate any API.
- Providing, via its modular structure, a way to create a basic system that provides common services, plus several subsystems built on this base and sharing the base, each providing support for a given API.
- Supporting multiple APIs running on the same system concurrently, in such a way that diverse applications can communicate transparently.

Support for JavaTM Applications

The ChorusOS operating system provides an execution environment that, when combined with a JavaTM Runtime Environment (JRE), supports real-time applications and Java applications running on the same machine, each in the appropriate environment.

Transparent Inter-Process Communication (IPC)

Based on industry standards, the Transparent Inter-Process Communication (IPC) facility of the ChorusOS operating system 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, the 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.

Operating System Components

Optional features are implemented as components that can be added to, or removed from, an instance of 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 is always present in an instance of the ChorusOS operating system. Optional components in the operating system provide the following services:

- Actor management
- Scheduling
- Memory management
- Hot restart and persistent memory
- Inter-thread communication
- Time management
- Inter-process communication
- Local Access Point (LAP)
- Tools support
- C_INIT
- File system options
- I/O management
- Networking
- Administration

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 to be called. Some library functions are independent of any specific component and are always available.

The following sections provide detailed descriptions of the various optional components of the operating system. Each component is identified by a name which is used by the configuration tools and within applications.

Table 1-1 shows the component groups.

TABLE 1-1 Operating System Optional Components

Component	Name
Actor management	
Dynamic actor loading management	ACTOR_EXTENDED _MNGT
User-mode extension support	USER_MODE
Dynamic libraries	DYNAMIC_LIB
Compressed file management	GZ_FILE
Scheduling	
POSIX round-robin scheduling class	ROUND_ROBIN
Memory management	
Virtual (user and supervisor) address space	VIRTUAL_ADDRESS_SPACE
On-demand paging	ON_DEMAND_PAGING
Hot restart and persistent memory	
Hot restart	HOT_RESTART
Inter-thread communication	
Semaphores	SEM
Event flag sets	EVENT
Mutual exclusion locks supporting thread priority inversion avoidance	RTMUTEX
Time management	
Periodic timers	TIMER
Thread and actor virtual timer	VTIMER
Date and time of day	DATE
Real-time clock	RTC

TABLE 1-1 Operating System Optional Components *(continued)*

Component	Name
Inter-process communication	
Location-transparent inter-process communication	IPC
Remote (inter-site) IPC support	IPC_REMOTE
Remote IPC communications medium	IPC_REMOTE_COMM
Mailbox-based communications mechanism	MIPC
POSIX 1-compliant message queues	POSIX_MQ
POSIX 1-compliant shared memory objects	POSIX_SHM
LAP	
Local name server for LAP binding	LAPBIND
LAP validity-check option	LAPSAFE
Tools support	
Message logging	LOG
Profiling and benchmark support	PERF
System monitoring	MON
System debugging	DEBUG_SYSTEM
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
UFS file system	UFS
I/O management	
Network packet filter	BPF
Swap support	FS_MAPPER
Driver for IDE disk	IDE_DISK
/dev/mem, /dev/kmem, /dev/null, /dev/zero	DEV_MEM
Support for RAM disk	RAM_DISK

TABLE 1-1 Operating System Optional Components *(continued)*

Component	Name
Support for FLASH media	FLASH
Virtual TTY	VTTY
Driver for SCSI disk	SCSI_DISK
Support for IPC	IOM_IPC
Support for OSI	IOM_OSI
Networking	
Serial link IP	SLIP
POSIX 1003.1g-compliant sockets	POSIX_SOCKETS
Point-to-point protocols	PPP
Local sockets and pipes	AF_LOCAL
Administration	
ChorusOS statistics	ADMIN_CHORUSSTAT
<code>ifconfig</code> administration command	ADMIN_IFCONFIG
<code>mount</code> administration command	ADMIN_MOUNT
<code>rarp</code> administration command	ADMIN_RARP
<code>route</code> administration command	ADMIN_ROUTE
<code>shutdown</code> administration command	ADMIN_SHUTDOWN
<code>netstat</code> administration command	ADMIN_NETSTAT

Not all these components are supported on all platforms. See the appropriate book in the *ChorusOS 4.0 Target Family Documentation Collection* for details of which components are available for your platform.

Some options are dependent on others. These dependencies are managed automatically by the configuration tools and it is not necessary to include prerequisite options explicitly. Some options are mutually exclusive, and the configuration tools will not permit you to include more than one option from a mutually exclusive set.

The Core Executive

The essential services required to support real-time applications are provided by the executive. The core executive can support multiple, multi-threaded applications running in both user and supervisor memory space. It provides the following kernel functionality:

- Support for multiple independent applications
- Support for user and system applications
- Support for applications in user and supervisor address space
- Dynamic memory management

See `CORE(5FEA)` for further details.

Optional Operating System Services

Actor Management

ACTOR_EXTENDED_MNGT

The `ACTOR_EXTENDED_MNGT` feature provides extended management functions for actors, including dynamic loading and control of actors. This feature also provides the underlying support for more advanced features such as support of dynamically loadable libraries (`DYNAMIC_LIB`) and uncompression of actors or libraries at load time (`GZ_FILE`).

For more details, see `ACTOR_EXTENDED_MNGT(5FEA)`.

USER_MODE

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

DYNAMIC_LIB

The `DYNAMIC_LIB` feature provides support for dynamic libraries within Sun Embedded Workshop. It requires the `ACTOR_EXTENDED_MNGT` feature, so that actors can be linked with dynamic libraries. 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.

For more details, see `DYNAMIC_LIB(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 requires the `ACTOR_EXTENDED_MNGT` feature.

For more details, see `GZ_FILE(5FEA)`.

Scheduling

A scheduler is a feature which 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 kernel internal interface. A scheduler is mandatory in all kernel instances. The core executive includes the default `FIFO` scheduler.

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

More detailed information about these scheduling classes is found in `threadScheduler(2K)`.

For more details on scheduling, see `SCHED(5FEA)`.

ROUND_ROBIN

The optional `ROUND_ROBIN` scheduler feature enables the additional `CLASS_RR` scheduling class, which is similar to `CLASS_FIFO` but adds round-robin time slicing based on a configurable time quantum.

For more details, see `ROUND_ROBIN(5FEA)`.

Memory Management

There are three memory management models, `MEM_FLAT`, `MEM_PROTECTED`, and `MEM_VIRTUAL`. The model used is determined by the settings of the `VIRTUAL_ADDRESS_SPACE` and `ON_DEMAND_PAGING` features. See `MEM(5FEA)` for more details.

■ `MEM_FLAT`

This memory management model provides simple memory allocation services. The kernel and all applications run in one unique unprotected address space.

■ `MEM_PROTECTED`

This memory management model is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

- `MEM_VIRTUAL`

This memory management model supports full virtual memory with swapping in and out on secondary devices. It has been specifically designed to implement distributed UNIX subsystems on top of the kernel.

`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 necessarily 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.

Hot Restart and Persistent Memory Management

The `HOT_RESTART` feature provides support for rapidly reloading and reinitializing failed ChorusOS operating system actors, without accessing stable storage. Actors which benefit from this support are known as *restartable actors*. `HOT_RESTART` also provides all actors (not just restartable actors) with a means of storing persistent data, data which can persist beyond the lifetime of a run-time instance of an actor.

The main services exported by the `HOT_RESTART` feature are:

- An actor restart mechanism which detects crashes in restartable actors, and automatically restarts them from an actor image in persistent memory.
- Persistent memory allocation. Actors can allocate blocks of persistent memory to store data which will persist beyond the actor's lifetime.

- A site restart mechanism to restart the kernel, boot actors and all restartable actors on a system without accessing stable storage.

For more details, see `HOT_RESTART(5FEA)`

Inter-thread communication

The ChorusOS operating system provides the following services to support multithreaded programming:

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 *pass*) and V (or *free*).

- 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 actors' address spaces. No kernel 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 may use is therefore unlimited.

For more details, see `SEM(5FEA)`.

EVENT

The `EVENT` feature provides the management of event flag sets.

An event flag set is a set of bits in memory that is associated with a thread wait queue. Each bit is associated with one event. Event flag sets are data structures allocated in the actors' address spaces. No kernel 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 may use is therefore unlimited.

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 on a conjunctive (*and*) or disjunctive (*or*) subset of the events in one event flags set. Several threads may be pending on the same event. In that case, each of the threads will be made eligible to run when the event occurs.

For more details, see `EVENT(5FEA)`.

RTMUTEX

The `RTMUTEX` feature provides mutual exclusion locks, using a priority inheritance protocol, in order to avoid thread priority inversion problems.

For more details, see `RTMUTEX(5FEA)`.

Time Management

The ChorusOS operating system provides the following time management features:

- Interrupt-level timing
- General interval timing
- Time of day (universal time)
- System time
- Thread execution timing
- Benchmark timing

The interrupt-level timing feature is always available and provides a traditional, one-shot time-out service. Time-outs and the time-out 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.

TIMER

The `TIMER` feature implements a high-level interval timing service for both user and supervisor actors. It includes one-shot and periodic timers. The time-out notification is achieved through user-provided handler threads which are woken up in the application actor. Handler threads may invoke any kernel or subsystem system call.

For more details, see `TIMER(5FEA)`.

VTIMER

The virtual time option provides a number of functions that are typically used by higher-level operating systems for controlling and accounting thread-execution.

Virtual time-outs can be set on:

- Individual threads, to support subsystem-level timers.
- Entire actors (that is, multiple threads), to support process CPU limits.

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

Execution accounting may be limited to execution within the home actor of the thread (internal execution time) or may be extended to include cross-actor invocations, such as system calls (total execution time).

For more details, see `VTIMER(5FEA)`.

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 kernel.

For more details, see `DATE(5FEA)`.

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 more information, see `RTC(5FEA)`.

Inter-process communication

The ChorusOS operating system provides Inter Process Communication (IPC), allowing threads to communicate and synchronize, even when they do not share the same memory space.

Communication is achieved by the exchange of messages through ports, and IPC supports port migration, whereby the messages sent to a given port can be transferred to a new process in a way that is transparent to the application.

The ChorusOS operating system also includes a mailbox (MIPC) mechanism that provides a shared communication environment for actors within an application.

IPC

The `IPC` feature provides powerful asynchronous and synchronous communication services.

The `IPC` feature exports the following basic communication abstractions:

- The unit of communication (`message`)
- Point-to-point communication endpoints (`port`)
- Multi-cast communication endpoints (`groups`)

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.

For more details, see `IPC(5FEA)`.

For information on how to configure `IPC` for local, Ethernet, and VME use, see Appendix A.

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 co-located on the same node.

For information on how to configure `IPC` for local, Ethernet, and VME use, see Appendix A.

IPC_REMOTE_COMM

If you set `IPC_REMOTE`, you can specify the communication method by setting the `IPC_REMOTE_COMM` feature. By default, this is set to `EXT` for external networking protocols. You can also set it to `VME`, and have the communication managed by the kernel directly.

For information on how to configure `IPC` for local, Ethernet, and VME use, see Appendix A.

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 in a very efficient way. In particular, supervisor and user actors of a same application can exchange messages with the MIPC service. Furthermore, these messages can be initially allocated and sent by interrupt handlers in order to be processed later in the context of threads.

See Chapter 8 for more information about using message spaces.

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

POSIX_MQ

The `POSIX_MQ` feature is a compatible implementation of the POSIX 1 real-time message queue API. For general information on this feature, see `intro(2POSIX)`, and the POSIX standard (IEEE Std 1003.1b-1993).

For more details, see `POSIX_MQ(5FEA)`.

POSIX_SHM

The `POSIX_SHM` feature is a compatible implementation of the POSIX 1 real-time shared memory objects API. For general information on this feature, see `intro(2POSIX)`, and the POSIX standard (IEEE Std 1003.1b-1993).

For more details, see `POSIX_SHM(5FEA)`.

LAP

Low overhead, same-site invocation of functions and APIs exported by supervisor actors may be done 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. In addition, optional extensions provide safe on-the-fly shutdown of local service routines and a local name binding service (see the `LAPSAFE` and `LAPBIND` features).

See `CORE(5FEA)` for further details.

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).

For more details, see `LAPBIND(5FEA)`.

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 more details, see `LAPSAFE(5FEA)`.

Tools support

The ChorusOS operating system provides the following support for debugging.

LOG

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

For more 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 more details see `PERF(5FEA)`.

MON

The MON feature provides a means to monitor the activity of kernel 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 more details see `MON(5FEA)`.

DEBUG_SYSTEM

The `DEBUG_SYSTEM` feature enables remote debugging of the ChorusOS operating system with the XRAY Debugger for ChorusOS. XRAY communicates with the ChorusOS debug server (see `chserver(1CC)`) through the RDBS protocol adapter (see `rdbbs(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 more details see `DEBUG_SYSTEM(5FEA)`.

C_INIT Options

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. `console` is usually run at the end of the `sysadm.ini` file. It can also be run through `rsh` if it is available.

See `C_INIT(1M)` for a detailed description of `console` and other `C_INIT` commands.

RSH

This feature gives access to `C_INIT` commands through the `rsh` service. When this feature is set, the `C_INIT rshd` command starts the `rsh` demon. `rshd` is usually run from the end of the `sysadm.ini` file. It can also be run from the local console if it is available.

See `C_INIT(1M)` for a detailed description of `rshd` and other `C_INIT` commands.

File System Options

The ChorusOS operating system supports the following types of file system:

- Network file system, NFS (client and server)
- MS-DOS file system
- UNIX file system, UFS

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 more details, see `FIFOFS(5FEA)`.

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, PPP or SLIP. The `NFS_CLIENT` requires the `POSIX_SOCKETS` feature to be configured.

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

NFS_SERVER

The `NFS_SERVER` feature provides the services to provide an NFS server on top of a local UFS file system. It provides only the server side implementation of the protocol, the client side being provided by the `NFS_CLIENT` feature. The `NFS_SERVER` requires `POSIX_SOCKETS` and `UFS`.

For more details, see `NFS_SERVER(5FEA)`.

MSDOSFS

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

At least one of `RAM_DISK`, `IDE_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.

For more details, see `MSDOSFS(5FEA)`.

UFS

The UNIX file system option provides support for a disk-based file system, that is, 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

For more details, see `UFS(5FEA)`.

I/O Management

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

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 `ADMIN_RARP` feature, or Dynamic Host Configuration Protocol client (`dhclient(1M)`).

For more details, see `BPF(5FEA)`.

FS_MAPPER

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

For more details, see `FS_MAPPER(5FEA)`.

IDE_DISK

The `IDE_DISK` feature provides an interface to access IDE disks. These disks may then be initialized and used as regular file systems. The `IDE_DISK` feature relies on the IDE bus support provided by the BSP to get access to disks connected on that bus.

For more details, see `IDE_DISK(5FEA)`.

DEV_MEM

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

For more details, see `DEV_MEM(5FEA)`.

RAM_DISK

The `RAM_DISK` feature provides an interface to chunks of memory which 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.

For more details, see `RAM_DISK(5FEA)`.

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 4.0 Target Family Documentation Collection* for details of which target family architecture supports the Flite 1.2 BSP.

For more details, see `FLASH(5FEA)`.

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 `SLIP` and `PPP` features.

For more details, see `VTTY(5FEA)`.

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.

For more details, see `SCSI_DISK(5FEA)`.

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 more details, see `IOM_IPC(5FEA)`.

IOM_OSI

The `IOM_OSI` feature provides support for the `ethOSISStackAttach(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 more details, see `IOM_OSI(5FEA)`.

Networking

The following features provide various methods of networking on the target:

SLIP

The `SLIP` feature allows serial lines to be used as network interfaces. This feature needs to be configured in order to fully support the `ADMIN_SLIP` feature as well as the various `slip` related commands provided by the Sun Embedded Workshop system.

For more details, see `SLIP(5FEA)`.

POSIX_SOCKETS

The `POSIX_SOCKETS` feature provides a TCP/IP stack through POSIX-compatible socket system calls. For general information on this feature, see `intro(2POSIX)` and the POSIX draft standard P1003.1g. However, `POSIX_SOCKETS` only provides support of the `AF_INET` domain. The `AF_LOCAL` domain support is provided by the `AF_LOCAL` feature.

For more details, see `POSIX_SOCKETS(5FEA)`.

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 in order to fully support the `ADMIN_PPP` feature as well as the various `PPP` related commands provided by the Sun Embedded Workshop system.

For more details, see `PPP(5FEA)`.

AF_LOCAL

The `AF_LOCAL` feature provides support for the `AF_LOCAL` domain for sockets. It requires and complements the `POSIX_SOCKETS` feature which provides the `AF_INET` domain independently.

For more details, see `AF_LOCAL(5FEA)`.

Administration

The ChorusOS operating system provides the following optional administration features:

ADMIN_CHORUSSTAT

The `ADMIN_CHORUSSTAT` feature provides support for the built-in `chorusStat` command of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will display an error message. This feature affects the content of the `ADMIN` system actor. For more information on the `chorusStat` service, refer to `chorusStat(1CC)`. Note that even if the `ADMIN_CHORUSSTAT` feature is not configured, you can get the ChorusOS operating system statistical information by running the `chorusStat` command, which is a stand-alone version of the built-in `C_INIT` command.

For more details, see `ADMIN_CHORUSSTAT(5FEA)`.

ADMIN_IFCONFIG

The `ADMIN_IFCONFIG` feature provides support for the built-in `ifconfig` command of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will display an error message. This feature affects the content of the `ADMIN` system actor. For more information on the `ifconfig` service, refer to `ifconfig(1M)`. Note that even if the `ADMIN_IFCONFIG` feature is not configured, you can configure network interface parameters by running the `ifconfig` command which is a stand-alone version of the built-in `C_INIT` command. However, in order to be able to set up the network interface of the target system appropriately at initialization time, the `ADMIN_IFCONFIG` feature is usually set.

For more details, see `ADMIN_IFCONFIG(5FEA)`.

ADMIN_MOUNT

The `ADMIN_MOUNT` feature provides support for the built-in `mount` and `umount` commands of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will display an error message. This feature affects the content of the `ADMIN` system actor. For more information on the `mount` service, refer to `mount(1M)`. This feature provides support to mount and unmount UFS, MS-DOS and NFS file systems. If this feature is not set, there will be no way to run a command to mount a file system within the target system. In this type of configuration, file systems will have to be mounted by user provided applications embedded within the boot image using the `mount(2POSIX)` system call.

For more details, see `ADMIN_MOUNT(5FEA)`.

ADMIN_RARP

The `ADMIN_RARP` feature provides support for the built-in `rarp` command of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will display an error message. This feature affects the content of the `ADMIN` system actor. For more accurate information on the `rarp` service, refer to `C_INIT(1M)`. The `ADMIN_RARP` feature enables the system to retrieve its local IP address using the RARP protocol, and to configure a network interface accordingly. This feature requires the `ADMIN_IFCONFIG` feature.

For more details, see `ADMIN_RARP(5FEA)`.

ADMIN_ROUTE

The `ADMIN_ROUTE` feature provides support for the built-in `route` command of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will display an error message. This feature affects the content of the `ADMIN` system actor. For more information on the `route` service, refer to `route(1M)`. Note that even if the `ADMIN_ROUTE` feature is not configured, you can still manage the routing tables of the Sun Embedded Workshop system by running the `route` command, which is a stand-alone version of the built-in `C_INIT` command. However, in order to be able to set up the routing tables of the target system appropriately at initialization time, the `ADMIN_ROUTE` feature is usually set.

For more details, see `ADMIN_ROUTE(5FEA)`.

ADMIN_SHUTDOWN

The `ADMIN_SHUTDOWN` feature provides support for the built-in `shutdown` and `reboot` commands of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` commands will display an error message. This feature affects the content of the `ADMIN` system actor. For more information on the `shutdown` service, refer to `shutdown(1M)`. This feature permits the stopping of all or part of the system, and possibly to reboot the system. Note that even if the `ADMIN_SHUTDOWN` feature is not configured, it may still be possible to stop the system by running the `shutdown` command, which is a stand-alone version of the built-in `C_INIT` command.

For more details, see `ADMIN_SHUTDOWN(5FEA)`.

ADMIN_NETSTAT

The `ADMIN_NETSTAT` feature provides support for the built-in `netstat` command of `C_INIT(1M)`. If the feature is not configured, the built-in `C_INIT` command will

display an error message. This feature affects the content of the ADMIN system actor. For more information on the `netstat` service, refer to `netstat(1CC)`. Note that even if the `ADMIN_NETSTAT` feature is not configured it may still be possible to get the network status by running the `netstat` command, which is a stand-alone version of the built-in `C_INIT` command.

For more details, see `ADMIN_NETSTAT(5FEA)`.

Configuring ChorusOS

The ChorusOS operating system provides two standard configuration profiles. These are useful starting points for defining your own configuration.

The 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 IPC over Ethernet and an 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.

The 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.

Development Environment Components

The development environment provided in Sun Embedded Workshop has the following major components:

- A C and C++ Development Toolchain, including the GNU `gcc` and `g++` cross-compilers, which are widely-recognized as amongst the best C and C++ compilers available on the market in terms of robustness, efficiency, and speed.

- A new debugging framework and a C and C++ reference debugger, Mentor Graphics Corporation *XRAY Debugger for ChorusOS*, which offers the following features:
 - Easy-to-use graphical user interface
 - Support for debugging several applications running on multiple targets with different processor architectures
 - Multithreaded user and supervisor applications, including relocatable ones, can be debugged
 - Flexible thread handling: one window per thread, breakpoint per thread or per application
 - The ChorusOS operating system abstractions related to debugged applications or global to the system can be visualized
 - Application debug over Ethernet or serial line, and system debug over a serial line.
- Configuration Tools: The ChorusOS operating system is configured simply by providing a list of the components that are required. Sun Embedded Workshop 4.0 includes a graphical tool, called *ews*, for configuring the system. This tool provides a user-friendly interface for configuring the ChorusOS operating system, and shows the dependencies between components. A command-line interface for configuration is also available. In addition to the ability to select only the components required for the operating system, Sun Embedded Workshop 4.0 supports three other levels of system configuration:
 - Resources. For the list of selected components, it is possible to fix the amount of resources to be managed, and to set the value of certain tunable parameters. For example, the amount of memory reserved for network buffers.
 - Boot Actors. It is possible to include additional actors in the memory image that are loaded at boot time.
 - Environment. System-wide configuration parameters can be fixed by setting environment strings, similar to environment variables used in UNIX systems, which the operating system and actors retrieve when they are initialized.
- A set of libraries:
 - Thread-safe C++
 - ANSI-C (POSIX 1003.1 compliant)
 - POSIX 1003.1-compliant timers, message queues, shared memory, semaphores, and pthreads
 - POSIX 1003.1-compliant I/O
 - POSIX 1003.1g-compliant sockets
 - Thread-safe mathematical ANSI-C
 - C++ iostream
 - C++ exceptions

- STL 3.1 (Standard Template Library)
- Management of per-thread private data
- X11, Xaw, Xext, Xmu, and Xt libraries
- Sun RPC

Debugging Architecture

This release of the ChorusOS operating system introduces an open, debugging architecture, as specified by the *ChorusOS Debug Architecture and API Specifications* document. The debug architecture relies on a host-resident server which abstracts the target platform to host tools, in particular debuggers. This API specification document is intended to be used by third parties who wish to implement their own debuggers for ChorusOS systems.

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

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 kernel, the BSP and the system protocol stacks.

For more details, see Chapter 10.

Management Utilities

Sun Embedded Workshop 4.0 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.

- *Bootmonitor* is used to boot the ChorusOS operating system remotely, by using `tftp`, when the target does not provide an embedded boot facility. This facility is not available on all targets.
- *Default Console* is used to direct all console I/O to a remote host over a serial line.
- *Remote Shell* is used to execute commands remotely on the target from the host. In particular, this feature allows applications to be loaded dynamically.
- *Resource Status* is used to list the current status of all operating system resources, for example, actors, threads, and memory.
- *Logging* (`LOG`) is used to log operating system events as they occur on the target.
- *Monitoring* (`MON`) is used to monitor operating system objects, so that user-defined routines are called when certain operations are performed, or certain events occur, on specified objects.
- *Profiling* is used to run profiling sessions on system applications.
- *Benchmarking* (`PERF`) is used to benchmark the operating system.

Development Lifecycle

This section provides an overview of the stages in using Sun Embedded Workshop to develop an application or system. It provides a high-level summary of the tasks described later in this book and elsewhere in the documentation set.

Installing Sun Embedded Workshop

Installing the Development Environment

The *ChorusOS 4.0 Installation Guide* explains how to download and install Sun Embedded Workshop.

When the installation is complete, you have all the binary components required to build an instance of the ChorusOS operating system. To create a system image, follow the instructions in the appropriate *ChorusOS 4.0 Target Family Documentation Collection*.

Installing a Boot Server

A boot server is a system that provides an instance of the ChorusOS operating system for downloading to target systems. A boot server is useful if you download

the same image to many targets. To install an instance of the ChorusOS operating system on a boot server, follow the instructions in the *ChorusOS 4.0 Installation Guide*. Note that the system where you installed the development environment can be used as a boot server.

Installing on a Target System

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

- Download the image at boot time from a boot server
- Load the image from media located on the target system

Developing an Application

Configuring the System

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 “Optional Operating System Services” on page 35 for information about optional components of the ChorusOS operating system. See Chapter 3 for information about configuring the ChorusOS operating system to include the components you require.

Writing an Application

Chapter 4 gives a summary of how to use Sun Embedded Workshop to create an application, including the following information:

- General principles of developing an application that runs on the ChorusOS operating system
- The APIs available
- How to build the application
- Different ways of running the application

Chapter 10 explains how to debug your application.

Tuning

When your application is written, you can create a performance profile for it, 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 Chapter 11 for more information.

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, see the *ChorusOS 4.0 Porting Guide*.
- For information about adding a device driver, see the *ChorusOS 4.0 Device Driver Framework Guide*.
- For information about developing applications to use the hot restart functionality of the ChorusOS operating system, see the *ChorusOS 4.0 Hot Restart Programmer's Guide*.
- For information about using the flash memory feature, see the *ChorusOS 4.0 Flash Guide*.
- For information about the organization of the source code and how to use it, see the *ChorusOS 4.0 Production Guide*.

PART II

Using ChorusOS

Using ChorusOS

This chapter introduces the basic principles of using the ChorusOS operating system.

The ChorusOS System Image

The ChorusOS operating system is supplied with two standard images:

- `kernonly`, which contains the kernel only and provides a minimal base for porting
- `chorus`, which contains a full system image allowing configuration of the whole feature set

Refer to the appropriate book in the *ChorusOS 4.0 Target Family Documentation Collection* for information about building the `kernonly` and `chorus` system images from the distribution.

Downloading the System Image

Follow the boot instructions specific to your target, as described in *ChorusOS 4.0 Installation Guide*. Messages similar to the following are displayed:

```
ChorusOS r4.0.0 for Intel x86 - Intel x86 PC/AT  
Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.
```

```
Kernel modules : CORE SCHED_FIFO SEM MIPC IPC_L MEM_PRM KDB TICK MON ENV \
ETIMER LOG LAPSAFE MUTEX EVENT UI DATE PERF TIMEOUT LAPBIND DKI
MEM: memory device 'sys_bank' vaddr 0x7bc43000 size 0x189000
```

```
[messages from IOM]
```

```
Copyright (c) 1992-1998 FreeBSD Inc.
```

```
Copyright (c) 1982, 1986, 1989, 1991, 1993
```

```
The Regents of the University of California. All rights reserved.
```

```
max disk buffer space = 0x10000
```

```
/rd: sun:ram--disk driver started
```

```
C_INIT: started
```

```
[ messages from C_INIT and other boot actors ]
```

Basic Environment

In the basic environment, application actors are loaded at boot time as part of the system image. These actors are also known as boot actors.

When the system boots, actors included in the system image are loaded. For each actor, a thread is created and starts running at the actor's program entry point.

Building an Application Actor

This section assumes that you have built a chorus or kernonly system image in the *build_dir* directory. This example will create a simple Hello World actor.

1. Create a working directory where the actor will reside.
2. In this working directory, create a file named *Imakefile* containing the following lines:

```
Depend(hello.c)
EmbeddedSupActorTarget(hello_s.r,hello.o,)
```

3. Create a file named *hello.c* containing your Hello World program, written in C. For example:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return(0);
}
```

4. Generate a Makefile to build the actor, by typing the following command:

```
% ChorusOSMkMf build_dir/Paths
```

See ChorusOSMkMf(1CC) for more information about creating a Makefile.

5. Build the dependencies:

```
% make depend
```

6. Build the application:

```
% make
```

Your directory will now contain a supervisor actor, `hello_s.r`.

Embedding your Actor in the System Image

The easiest way to add the actor to the system image is to use the graphical configuration tool, `ews`. See “Adding an Actor to the ChorusOS System Image” on page 81 for a step-by-step guide on how to do this.

Alternatively, you can modify `conf/mkimage/applications.xml` so that it contains the list of applications that will be included in your archive. For example, to include your supervisor actor, `hello`, the content should be as follows:

```
<folder name='Applications' visible='yes'>
  <description>Placeholder for customer applications</description>

  <definition name='hello' configurable='yes'>
    <description>simple hello actor, in supervisor mode</description>
    <type name='File' />
    <value field='path'>
      <vstring>absolute_path_to_my_actor/hello_s.r</vstring>
    </value>
    <value field='bank'><ref name='sys_bank' /></value>
    <value field='binary'><ref name='supervisor_actor_model' /></value>
  </definition>

  <definition name='application_files' configurable='yes'>
    <description>application system image files</description>
    <condition>
      <or>
        <equal><var name='SYSTEM' /><const>chorus</const></equal>
```

```

        <equal><var name='SYSTEM' /><const>kernonly</const></equal>
    </or>
</condition>
<type name='FileList' />
<value index='size'><ref name='hello' /> </value>
</definition>

</folder>

```

Rebuild the system image using one of the following commands:

- If you want to build a kernel-only system, type:

```
% make kernonly
```

- If you want to build a complete chorus system, type:

```
% make chorus
```

- If you want to rebuild the system that you have previously built, type:

```
% make build
```

Running your Actor in the Basic Environment

Boot the system you have created on the target system. For detailed instructions, see the appropriate book in the *ChorusOS 4.0 Target Family Documentation Collection*.

After the system boots, the following message is displayed on the console:

```
'Hello World!'' ' ' '
```

Extended Environment

The extended environment is provided in the ChorusOS 4.0 release and comes with a special actor called `C_INIT` which is dedicated to administrative commands.

Within the extended environment, application actors can either be loaded at boot time, as described in the previous section, or dynamically using the `C_INIT` loading facility. Dynamic loading of actors is described in “Running the “Hello World” Example” on page 65.

The `conf/sysadm.ini` file is used to specify system initialization commands. Each entry of this file is a command to be executed by `C_INIT` during the kernel boot. Typical operations in `sysadm.ini` are network configuration, device initialization, file system mount. See “System Administration in the Extended Environment” on page 67 for details.

The `sysadm.ini` file is not accessed remotely at boot time but is included in the system image.

Communicating with the Target Using `rsh`

When the ChorusOS operating system image including the `RSH` feature is booted on the target machine, the `C_INIT` daemon interprets the commands sent from the host through `rsh` (see the `rshd` manpage on your host). For example, to list the options available, type:

```
% rsh target help
```

The following information is displayed by the `C_INIT` actor:

For details of these commands, see `C_INIT(1M)`.

Mounting the Host File System

The NFS root file system to be mounted on the target is generated in the ChorusOS operating system build directory by the command:

```
% make root
```

This command populates the build directory with the `root` directory that contains binary and configuration files to be accessed by the target system.

At start-up, the `C_INIT` daemon reads the `sysadm.ini` configuration file and executes all the commands. See `sysadm.ini(4CC)` for more information. This configuration file may contain instructions to mount the root file system. For example:

```
% mount hostaddr:chorus_root_directory /
```

If there are no root file system mount instructions in your `sysadm.ini` file, you must mount the root file system explicitly from the shell:

```
% rsh target mount hostaddr:chorus_root_directory /
```

where *target* is the name of the target, or its IP address, *hostaddr* is the IP address of the NFS host in decimal form (for example 192.82.231.1), and *chorus_root_directory* is the path of the target root directory on the NFS host (for example /home/chorus/root).

When the mount of the root file system is successful, the `C_INIT` daemon displays, for example, the following message:

```
C_INIT: 192.82.231.1:/home/chorus/root mounted as root file
system
```

The next message from `C_INIT` depends on whether the `/etc/security` file exists in the target root directory `/home/chorus/root`. If `/etc/security` exists, `C_INIT` displays:

```
C_INIT: system in secured mode
```

If `/etc/security` does not exist, `C_INIT` displays:

```
C_INIT: notice - system not in secured mode
```

You can check that the root file system is mounted using:

```
% rsh target mount
```

Make sure that the file system containing the `/home/chorus/root` directory can be accessed by NFS from the remote ChorusOS target.

Security

The `C_INIT` daemon authenticates users issuing commands from the host.

The ChorusOS operating system can be configured in secure mode, where remote host access is checked through the `/etc/security` administration file, located on the target root file system (see `security(4CC)`). In addition, users' credentials may be specified in this file, overriding default `C_INIT` configuration values.

If an `/etc/security` file exists, it must have read permissions for everybody to allow `C_INIT` to read it with the default credentials (user identifier 0 and group identifier 0). Secure mode will then be activated. In this mode, `C_INIT` authenticates every command it receives from the host. Authentication will fail for two reasons:

- The user name of the remote user which issued the `rsh` command is not found in the security file.
- The remote host from which the `rsh` command came is not in the remote host's list of users.

In this case, a permission denied message is sent back to the host and the command is aborted.

If the authentication procedure succeeds, the user's privilege credentials (user identifier or *uid*, group identifier or *gid* and additional groups) are read from the security file. Trusted users have access to the full set of `C_INIT` commands.

In non-secured mode, every user is treated as a trusted user and inherits the `C_INIT` default credentials (*uid* 0 and *gid* 0). In this case, if the host machine has exported the file system to be mounted with the default mapping of *root* to *nobody*, it is necessary that read and execute permissions for the target executable files be given to everybody. Otherwise `C_INIT` will not have the right to execute the application binaries.

Another way to circumvent this problem is by inhibiting that mapping of *root* to *nobody* on the host. Please consult your system administrator about this.

Running the “Hello World” Example

- Copy your executable application files into the *chorus_root_directory/bin* directory.

```
% cp hello_s.r chorus_root_directory/bin
```

This step is important as the applications must be in a directory on the host that is exported to the target system.

- To start the *hello* supervisor actor:

```
% rsh target arun /bin/hello_s.r
```

The *arun* command returns the actor identifier (*aid*) of the new actor:

```
Started aid = 13  
Hello World!
```

- To list the actors running on the target:

```
% rsh target aps
```

- To kill the actor, the id of which is *aid*:

```
% rsh target akill aid
```

- To display information about current memory usage:

```
% rsh target memstat
```

The ChorusOS operating system actors are loaded and locked in memory when they start. This means that physical memory for the actor's text, data and stack must be available at load time. The `memstat` command of `C_INIT(1M)` can be used to check whether enough physical memory is available on the target system.

Input/Output Management

When 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. Note that in the basic environment this API is the only one available.

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.

Any extended actor has access to two special files `/dev/console` and `/dev/null`. `/dev/console` always refers to the system console of the target.

Note that `select(2POSIX)`, `stat(2POSIX)`, and `fstat(2POSIX)` are not supported on the `/dev/console` and `/dev/null` devices, and there is no `tty` line discipline management for these devices.

System Administration in the Extended Environment

C_INIT Actor

In the extended environment, a special actor called `C_INIT` provides administrative commands for the following:

- network configuration, such as defining IP addresses and initializing network interfaces
- file system management, such as partitioning a disk and mounting a file system
- device management, such as binding a high level service (file system, networking, tty management) to an actual device driver

Here are the most frequently used `C_INIT` commands:

- `mknod`: defines special device files
- `mkdev`: binds high level services to an instance of a device driver
- `mount`, `umount`: mounts and unmounts file systems
- `arun`: launches executables
- `ifconfig`: defines IP addresses
- `route`, `rarp`, `netstat`, `ppp`, `ping`: miscellaneous networking commands
- `memstat`, `chorusStat`: prints system statistics
- `setenv`, `unsetenv`, `echo`, `help`, `sleep`, `reboot`, `shutdown`: miscellaneous system commands
- `rshd`, `console`, `source`: specifies the device from which commands can be accepted:
 - `rshd`: from a host through `rsh`
 - `console`: from system console
 - `source`: from a file

See `C_INIT(1M)` for a complete description.

These commands are invoked at system start-up, described in the following section, and later during the life of the system. During the life of the system, the `C_INIT` actor executes commands from the system console, or from a remote host through `rsh`.

System Start-up

At system start-up, the C_INIT actor executes the following steps:

1. sets up an initial virtual file system
2. executes commands from the configuration file `sysadm.ini`
3. executes commands from `/etc/rc.chorus` when a root file system is mounted (see C_INIT(1M))

Note - If the target has a valid IP address, the file `/etc/rc.chorus.<ip_address>` (if it exists) will be selected instead of `/etc/rc.chorus`. `<ip_address>` must be written in the usual dot notation, for example: 192.82.231.1.

The initial virtual file system in step 1 contains only two directories, `/dev` and `/image/sys_bank`. The `/dev` directory, initially empty, is used for the definition of special devices, like `/dev/tty01`. The `/image/sys_bank` directory contains all the components in the boot image:

- system actors such as `am`, `iom`, `C_INIT` and drivers
- system configuration files (`sysadm.ini`)
- user defined configuration files and executables

All of these components can be accessed like the files in an ordinary file system, using their path, for example: `/image/sys_bank/sysadm.ini`.

Note - To access `/dev` and `/image` directories on the virtual file system, `dev` and `image` directories must be present on your root file system, and this root file system must be mounted.

In step 2, the C_INIT actor executes commands from a configuration file called `sysadm.ini`. This file contains all the commands needed for the initial administration of the system, including networking, file system management and device management.

The `sysadm.ini` file can be customized. On the host, it is located in the `conf` subdirectory of the ChorusOS build directory. This file is automatically embedded in the boot image, in the `/image/sys_bank/sysadm.ini` file of the initial file system. This allows you to configure embedded targets which do not have access to a local or remote file system.

Initialization Examples

Below are typical commands of the `sysadm.ini` file.

- Associate ifnet interface 0 to a specific Ethernet driver:

```
% mkdev ifeth 0 /pci/epic/epic100
```

The pathname is optional. For more information, refer to `mkdev(1M)`.

Note - In the ChorusOS operating system, hardware devices are identified by a path in a device tree; the `mkdev` command connects to the driver instance servicing the indicated hardware device.

- Associate ifnet interface 0 to the first Ethernet driver found:

```
% mkdev ifeth 0
```

- Define the IP address of ifnet interface 0:

```
% ifconfig ifeth0 ip-address netmask ip-mask broadcast broadcast-addr
```

- Define the IP address using the rarp protocol on ifnet interface 0:

```
% rarp ifeth 0
```

- Associate a special device to a serial line driver:

```
% mknod /dev/tty01 c 0 0
% mkdev tty 0 /pci/pci-isa/ns16550-2
```

The third argument to `mknod`, 0, is the major device number identifying the serial line driver. The fourth argument to `mknod`, 0, is the minor device number identifying the hardware device at the POSIX level.

- Mount a local file system by defining required devices, then mount the disk:

```
% mknod /dev/sd0a b 10 0
% mknod /dev/rsd0a c 9 0
% mount /dev/sd0a /
```

See also “Automated File System Initialization” in the *ChorusOS 4.0 File System Administration Guide*.

- Mount a remote file system:

```
% mount host-ip-addr:host-path /
```

Configuring and Tuning

This chapter explains how to configure and tune a ChorusOS operating system.

- “Configuration Options” on page 71 explains what items can be configured and how they are defined.
- “Configuration Tools” on page 76 explains how to configure your system.

The ChorusOS operating system offers a high degree of flexibility, allowing you to tailor the system configuration to the requirements of your application. Depending on the system configuration, applications are offered a range of Application Programming Interfaces (APIs), and a range of development environment tools. Two standard configuration profiles are included in this ChorusOS operating system delivery: a standard configuration profile and an extended configuration profile. You can use one of these configuration profiles as the starting point for configuring your ChorusOS operating system.

Configuration Options

Configuring a ChorusOS operating system means defining all the components, and their characteristics, which are assembled to form a system image. There are several types of configuration options:

- Feature options: the ChorusOS operating system features
- Static tunable parameters
- Dynamic tunable parameters (the environment)
- System image components: system and application actors which are loaded at system boot time

Configuration settings, including the configuration profile definitions are stored in the configuration directory, `conf`, in your system image build area. The configuration directory is read and updated by both the command-line and graphical configuration tools.

Feature Options

A ChorusOS feature is a boolean variable, whose value determines whether or not a particular component is included in the system image. Setting a feature to `true` results in code being added to the kernel, providing additional services such as file system handlers, or networking protocols.

Feature options within the ChorusOS operating system are listed in “Operating System Components” on page 31.

Configuration Profiles

The ChorusOS operating system provides profiles which are used to set up an initial configuration. These profiles include or remove certain features in the system.

Two pre-defined configuration profiles, the `basic` profile and the `extended` profile, are provided to help you select an initial configuration for the ChorusOS operating system. The `extended` profile is the default profile, and does not need to be explicitly specified.

The *extended configuration profile* corresponds to a reference configuration for telecommunications systems. It includes support for networking using remote IPC over Ethernet and an NFS client. This uses the protected memory model.

The *basic configuration profile* corresponds to a realistic configuration, keeping the footprint small. With this configuration, applications are usually embedded in the system image and launched either at boot time or subsequently from the image file system or the boot file system. This configuration uses the flat memory model, to minimize the footprint. System administration is local, with `C_INIT` access through the console.

Table 3–1 shows the settings of all the features in the `extended` and `basic` configuration profiles.

TABLE 3–1 Feature settings in the `extended` and `basic` configuration profiles

Name	extended profile value	basic profile value
Kernel features		
<code>USER_MODE</code>	<code>true</code>	<code>true</code>

TABLE 3-1 Feature settings in the extended and basic configuration profiles *(continued)*

Name	extended profile value	basic profile value
VIRTUAL_ADDRESS_SPACE	true	false
SEM	true	true
EVENT	true	true
MONITOR	false	false
TIMER	true	true
DATE	true	true
RTC	true	true
PERF	true	true
IPC	true	true
LOG	true	true
MON	true	false
MIPC	true	true
LAPBIND	true	true
LAPSAFE	true	true
C_INIT features		
LOCAL_CONSOLE	false	false
RSH	true	false
IOM features		
AF_LOCAL	true	true
BPF	true	false
DEV_MEM	true	false
MSDOSFS	true	true
NFS_CLIENT	true	false
POSIX_SOCKETS	true	true
RAM_DISK	true	true
AM features		
ACTOR_EXTENDED_MNGT	true	true
ADMIN features		
ADMIN_IFCONFIG	true	true

TABLE 3-1 Feature settings in the extended and basic configuration profiles *(continued)*

Name	extended profile value	basic profile value
ADMIN_MOUNT	true	true
ADMIN_RARP	true	false
ADMIN_ROUTE	true	true
ADMIN_SHUTDOWN	true	true

Note - The MONITOR feature is an internal feature which is only used by the Java Virtual Machine.

Both configuration profiles include support for system debugging.

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 “Command-line Configuration Tool” on page 86). Once you have created your initial configuration, you can also use the graphical configuration tool `ews` (see “Graphical Configuration Tool” on page 76) to manage the configuration.

Tunable Parameters

Tunable parameters are system parameters which affect system behavior and capabilities. They are used to configure the kernel and the included features, to change their behavior, and adapt them to your needs. Typical examples of tunables are: maximum number of kernel objects, scheduler type and attributes for threads, or system clock frequency. Each system component or feature defines a number of these tunable parameters.

Static Parameters

Static parameters are tunable parameters whose values are permanently set within a system image. Changing these values requires rebuilding the system image.

The procedure for assigning new values to tunable parameters is detailed in “Changing Tunable Parameter Values” on page 89.

Dynamic Parameters

For some tunable parameters, an additional flexibility is offered: the ability to assign values to these parameters at various stages of system production and execution.

These types of parameters are called dynamic parameters. These dynamic parameters define the system environment.

Dynamic parameters form a system-wide environment. A basic set of services allows this environment to be constructed and consulted within a system image, at boot time and runtime.

Compared to static parameters, dynamic parameters require additional target data memory in order to store their names and values.

The procedure for modifying dynamic parameters is detailed in “Modifying the System Environment” on page 90.

System Image Components

The system image contains a configured version of the ChorusOS operating system, and possibly some user-defined applications (actors).

Depending on its configuration options, the ChorusOS operating system is itself built from a kernel and a collection of actors. These actors, which contribute to the implementation of some ChorusOS operating system features, are called ChorusOS operating system actors.

Configuration options concerning the system image components deal mainly with the inclusion of system and application actors within system images.

Configuration Files

The ChorusOS operating system configuration is expressed in ECML, an XML based language. There are several levels of configuration files, all located in the `conf` directory used to build the system image.

- `ChorusOS.xml` is the top level configuration file. The entire ChorusOS operating system configuration is accessible through this file, which contains references to all other configuration files.
- `mkconfig` is the directory containing the configuration information for each system component. Most of the information it contains relates to feature options and tunable parameters. For example:
 - `mkconfig/kern.xml` contains the kernel feature definitions and dependencies, and contains the tunables for the kernel. This file also contains default values for the standard configuration.
 - `mkconfig/kern_action.xml` contains specific configuration actions, including the production rules used internally for the configuration.

- `mkconfig/kern_f.xml` and `mkconfig/kern_action_f.xml` are additional configuration files identified by `_f.xml` that can be used to manage family-specific configuration options.
- `mkimage` is the directory containing all the information related to the system image build:
 - `mkimage/mkimage.xml` contains two configurable declarations:
 - `BOOT_MODE` is set to `ram` to build an image for RAM, or `rom` to build an image for ROM.
 - `SYSTEM` is set to `chorus` to build a default system image, or `kernonly` to build a kernel-only system image. Other system images are also available.
 - `mkimage/family.xml` contains the family dependent definitions.
 - `mkimage/model.xml` contains the binary models for the executable files.
 - `mkimage/target.xml` contains all configuration options related to the BSP, and also the list of drivers.
 - `mkimage/system.xml` contains all system binaries and the configuration of the system image.
 - `mkimage/applications.xml` describes the applications to be included in the chorus or kernonly system image.
- `basic` and `extended` are the two configuration profiles.

Configuration Tools

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

You can use either a graphical interface or a command-line interface to view and modify the characteristics of a ChorusOS operating system image.

Graphical Configuration Tool

The graphical configuration tool, `ews`, requires Sun Java JDK 1.2 (JAVA 2) to be installed and the location of the Java virtual machine to be in your path.

To start `ews` and open an existing configuration file, type:

```
$ ews -c config-file
```

The optional *config-file* specifies the path of the ChorusOS operating system configuration file `conf/ChorusOS.xml` to open at start-up.

To start `ews` without opening a file, type:

```
$ ews
```

User Interface Overview

When started, `ews` opens a main window, containing a menu bar and toolbar at the top, a navigation tree pane on the left, and an output view pane at the bottom. The rest of the window is occupied by a Multiple Document Interface (MDI) area, which is used to display other windows, like the `Properties Inspector`, or the `Find View` (both are described later). These other windows can be resized, moved, or closed just like any other window, but are constrained within the MDI area, and cannot be moved outside.

A screenshot of `ews` is shown in Figure 3–1.

The `Show Children View` window displays the sub-element, or first-level child, of a selected element.

The `Find` window is used to locate an element in the project view tree. Any element can be searched for by specifying a substring of its name or its type. The search can take place from the root, on the entire tree hierarchy, or from the selected element.

Configuring a ChorusOS Operating System Image

Open a Configuration File

The first operation is to open a ChorusOS configuration file (unless the `-c` option was used on the command line). For this, select the `Open` option in the `File` menu. A file selection dialog appears to select the configuration file to open. The configuration to open is the `conf/ChorusOS.xml` file located in the configuration directory. Once opened, a new configuration item is added to the navigation tree.

Note - More than one configuration may be opened in `ews` at the same time.

Browse the Configuration Tree

It is possible to browse the configuration by opening the elements in the navigation tree. There are two general kinds of elements in the tree: folders and variables. Folders are used to organize the configuration variables into hierarchical groups. A folder contains child elements that can be variables or folders. Variables are values used to configure the ChorusOS operating system image.

Disabled Elements

Some of the elements in the configuration tree may be grayed-out and cannot be edited. It is still possible to browse them, however. For example, some variables may depend on the presence of a specific feature: if this feature is not selected, and its current value is set to `false`, the corresponding tunables will be disabled.

Disabling of elements in the configuration is controlled by a condition. This is an optional property attached to some elements, and if the condition is evaluated to false, the element is disabled (elements without a condition property are always enabled). If a folder is disabled, all its child elements (folders and variables) are also disabled.

Invalid Elements

A configuration is invalid if there are one or more invalid elements in the configuration tree.

Configuring the Features and Tunables

The features of the ChorusOS operating system image are located in the various `Features` and `Tunables` folders. Features are expressed as boolean variables, and tunables are expressed as either integers or enumerated variables. The following properties for a tunable are visible in the `Properties Inspector`:

- Name
- Type
- Default value
- Current value

To change the value of a tunable parameter, edit the `Current value` property.

Setting a ChorusOS operating system Environment Variable

The values of the ChorusOS operating system environment variables are contained in the `env` variable located in the `Environment` folder. The `env` variable is a list, where each element represents an environment variable. This list may be empty for a new configuration. It is possible to add, remove or modify environment variables stored in this list.

Adding an Environment Variable

Select the `env` variable, right-click to display its context menu, and select `New Element`. The newly created variable is appended to the list (you might need to expand the list to see the new variable). Set the value of the new variable by editing its `value` field.

Modifying the Value of an Environment Variable

An environment variable is a structured variable containing two fields: a name and a value. The `name` field stores the name of the environment variable, and the `value` field stores the value of this environment variable. Edit the `value` field to change the value of the environment variable.

Deleting an Environment Variable

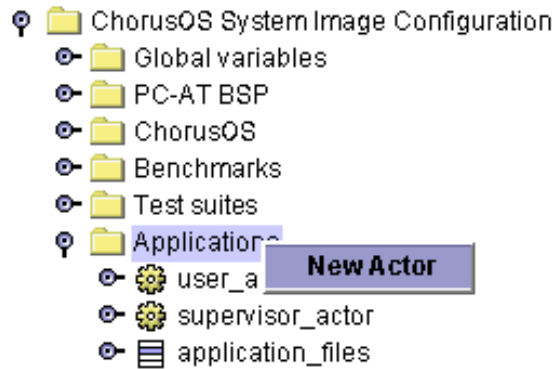
From the context menu of the environment variable, choose `Delete`.

Adding an Actor to the ChorusOS System Image

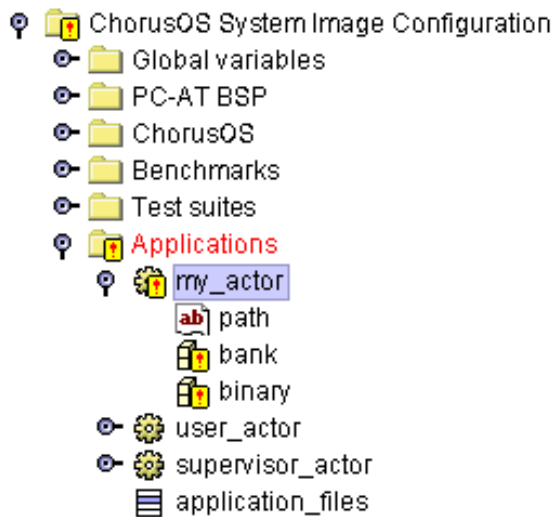
There are two stages to adding an actor to the system image:

1. Specify the new actor characteristics.

Open the `Applications` folder in the `ChorusOS System Image Configuration` folder. A newly-created `System Image Configuration` folder contains two templates for defining actors, one for user actors (`user_actor`) and one for supervisor actors (`supervisor_actor`). To create your actor definition, either modify or duplicate one of these templates, or choose `New Actor` from the context menu of the `Applications` folder:



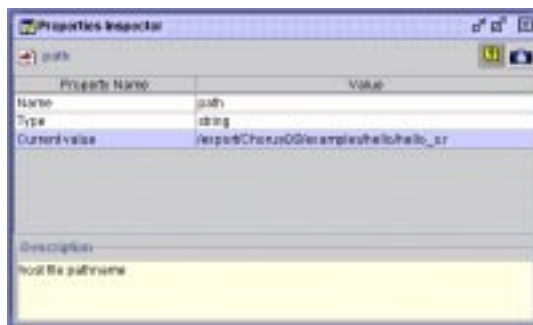
A new actor called `my_actor` is created. Click on the handle icon to the left of the actor, or double-click on `my_actor` itself, to reveal a list of fields, or children:



Invalid elements are indicated by an exclamation mark (!) over the icon. Your new actor is invalid because its field values are empty. Double-click on the `path` field to open the Properties Inspector window within the MDI:



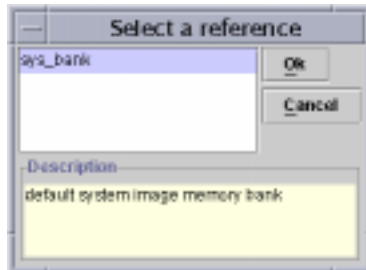
Enter the absolute pathname of your actor by double-clicking in the Value field of the Current Value property. For example:



Now double-click on the bank property to open up its Properties Inspector window, then double-click in the Value field of the Reference property. An ellipsis (. . .) will appear at the right hand side of the field:



Click on the ellipsis to open the reference selecting window, Select a reference window:

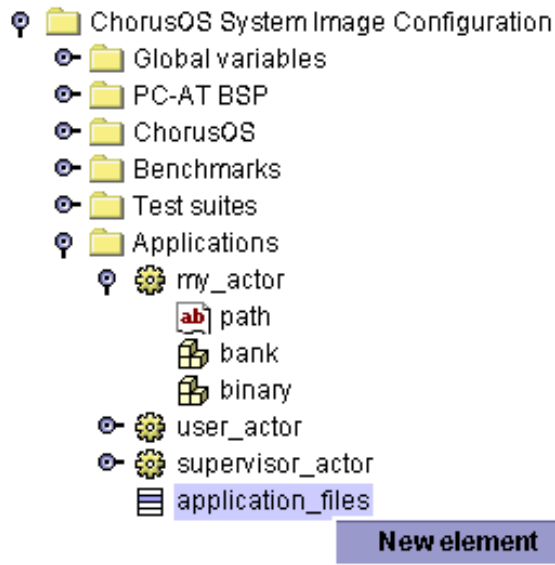


Click on the required reference, `sys_bank`, then click on `Ok`.

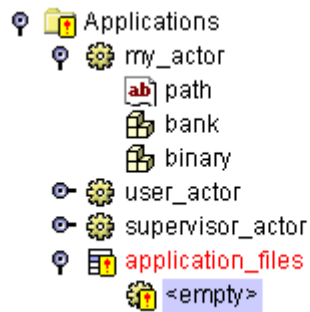
Now double-click on the binary property and perform similar actions to those you performed for the bank property.

2. Add the actor to the list of application files present in the system image.

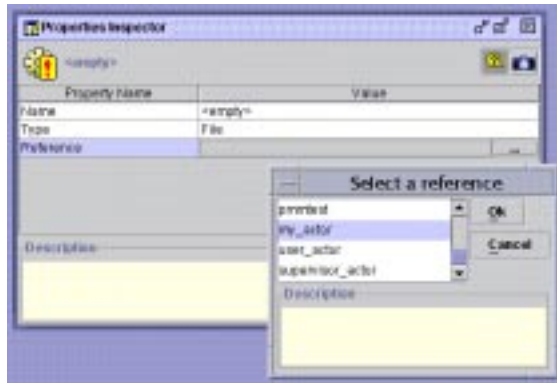
The `application_files` list in the ChorusOS System Image Configuration folder contains references to the actors that will be present in the ChorusOS operating system image. If an actor is defined but not referenced in this list, it will not be added to the image. Add your actor to this list choosing `New element` from its context menu:



An empty element will appear:



Update the element by opening it in the Properties Inspector and changing the Value field of the Reference property. Scroll down and select your newly defined actor, my_actor in this example, from the opened Select a reference window:



Click on Ok to complete the operation.

Note - Drivers, defined in the BSP folder of the ChorusOS System Image Configuration folder, may be added to the system image in exactly the same way.

Saving the Modified Configuration

After a configuration has been edited, it can be saved. For this, select the ChorusOS configuration item in the navigation tree (this is the root element of a configuration), and use its context menu. It is also possible to save it using the Save option in the File menu on the main menu bar, or the Save button on the toolbar.

Note - A modified configuration is displayed in red, as a visual warning that the file has changed.

Build the system image, as described in the next section.

Rebuilding the System Image

To rebuild the system image, select the ChorusOS configuration item in the navigation tree, and use the build item in its context menu (or the corresponding toolbar button). If the configuration file has not been saved since it was last modified, the tool will propose saving it, as the configuration needs to be saved in order to be built. If the configuration is invalid, it is not possible to build the corresponding ChorusOS operating system image.

During the build of the system image, various messages generated by the make tools are displayed in the output window.

It is possible to interrupt the build using the stop button on the toolbar. In this case, the system image is not built.

Command-line Configuration Tool


The following sections explain how to use the command-line configuration tool, `configurator`, for some common tasks.

Displaying the Configuration

The `configurator` utility provides an option to display the ChorusOS operating system configuration in HTML format. Within your build directory, type the following command:

```
$ configurator -display /tmp/ChorusOS.html
```

You can then use your browser to navigate through the `ChorusOS.html` file generated by this command.



```
o folder Kernel Configuration
  o corelinkconfig/ksen_1.xml
o folder Core Executive
  o feature HOT_RESTART
    o description: Hot restart support
    o type: bool
    o value: false
  o feature USER_MODE
    o description: User mode execution support
    o type: bool
    o value: true
  o feature ROUND_ROBIN
    o description: POSIX round-robin scheduling class
    o type: bool
    o value: false
o folder Kernel core tunables
  o tunable chorusSiteId
    o description: Unique Chorus Site Identifier. Needs only
      to be set if not automatically provided to the kernel by
      the board-specific boot. WARNING: when set, must be
      set with different values within every boot image
      dedicated to different target boards
    o type: int
    o value: 0
```

Figure 3-2 Kernel Configuration Displayed in HTML

Selecting a Configuration Profile

Two predefined profiles are provided, as described in “Feature Options” on page 72. To select the basic profile, type:

```
% configurator -p conf/basic
```

To re-select the extended (default) profile, type:

```
% configurator -p conf/extended
```

Adding, Removing, or Listing a Feature

You can use the `configurator` utility to add, remove, or list a feature.

Adding a Feature

To add a feature, type:

```
% configurator -set feature_name=true
```

The status of the `feature_name` is set to `true`.

For example, to add the `EVENT` feature to the default configuration:

```
% configurator -set EVENT=true
```

The `configurator` utility does not handle feature dependencies automatically. If you define a set that is not complete, an error message is displayed.

Removing a Feature

To remove a feature, type:

```
% configurator -set feature_name=false
```

The status of the `feature_name` feature is set to `false`.

For example, to remove the `EVENT` feature:

```
% configurator -set EVENT=false
```

You can reset the value of a feature to the default as follows:

```
% configurator -reset EVENT
```

Note - The `reset` command resets the value to the extended profile default.

Listing a Feature

You can check the value of a feature as follows:

```
% configurator -list feature feature_name
```

The output lists the feature and its status. If you omit *feature_name*, all features are displayed:

```
% configurator -list feature

SEM bool 'true'
EVENT bool 'true'
RTMUTEX bool 'false'
TIMER bool 'true'
VTIMER bool 'false'
DATE bool 'true'
```

You can list a feature in greater detail with the `-info` option:

```
% configurator -info feature feature_name
```

The output lists the feature, its status, possible values and its description. For example:

```
% configurator -info feature NFS_SERVER

NFS_SERVER:bool='false'
Possible values: true|false
Description: NFS server access from target machine
```


Changing Tunable Parameter Values

Tunable parameters are defined by symbolic names within the ChorusOS operating system components. Symbolic names include dots (.), to maintain compatibility with previous releases of the ChorusOS operating system.

The definition of a tunable parameter includes the definition of a default value for this parameter. Integer values of tunable parameters are expressed either as integers or as hexadecimal numbers.

To change the value of a tunable parameter, use:

```
% configurator -set tunable_name=value
```

For example, to re-configure the kernel to allow the creation of 300 threads:

```
% configurator -set kern.exec.maxThreadNumber=300
```

You can check the value of a tunable parameter as follows:

```
% configurator -list tunable tunable_name
```

You can list the values of all the kernel executive tunables as follows:

```
% configurator -list tunable kern.exec.*
```

The output lists the kernel executive tunables and their values:

```
kern.exec.maxCpuNumber int '1'  
kern.exec.maxActorNumber int '64'  
kern.exec.maxThreadNumber int '300'  
kern.exec.bgStackSize int '0x1000'  
kern.exec.dflSysStackSize int '0x3000'  
kern.exec.dflUsrStackSize int '0x4000'  
kern.exec.dblFltStackSize int '0x800'  
kern.exec.intrStackSize int '0x3000'
```

You can list a tunable parameter in greater detail with the `-info` option:

```
% configurator -info tunable tunable_name
```

The tunable, its value and its description are displayed:

```
% configurator -info tunable kern.lap.*  
  
kern.lap.maxLapBindNumber:int='256'  
Description: Maximum number of bind LAPs  
  
kern.lap.maxLapSafeNumber:int='128'  
Description: Maximum number of safe LAPs
```

Modifying the System Environment

The system environment is defined by the set of dynamic parameters. The system environment is a set of name-value pairs, where `name` and `value` are character strings. Values for system environment variables can be obtained by the system and applications at runtime using the `sysGetEnv(2K)` system call.

To display all the system environment variables, type:

```
% configurator -list env
```

To set a new environment variable, or change its value:

```
% configurator -setenv envar=value
```

Here is an example:

```
% configurator -setenv MESSAGE='HelloWorld'
```

To unset a variable, so that it is removed from the environment, type:

```
% configurator -resetenv envar
```

Rebuilding the System Image

After you have finished modifying the configuration, rebuild the system image by typing:

```
% make build
```


PART III Programming Overview

Programming Overview

This chapter introduces the steps involved in developing applications, also called actors, that run on the ChorusOS operating system. It includes the following sections:

- “ChorusOS Applications” on page 96 is a summary of the general principles of developing an application that runs on the ChorusOS operating system.
- “Application Programming Interfaces” on page 98 contains a summary of the APIs available.
- “Developing ChorusOS Applications” on page 103 explains how to build a component to be included in the ChorusOS system image.
- “Using Dynamic Libraries” on page 109 presents the two types of library in the ChorusOS operating system, and how to use them.

System development and advanced programming topics are not covered.

- For information about porting ChorusOS software to another target, see the *ChorusOS 4.0 Porting Guide*.
- For information about adding a device driver, see the *ChorusOS 4.0 Device Driver Framework Guide*.
- For information about the tools used to build ChorusOS, see the *ChorusOS 4.0 Production Guide*.

Note - The source code for many of the examples shown in this book is provided in the examples directory. By default this directory is

`/opt/SUNWconn/SEW/4.0/chorus-<target>/src/opt/examples.`

ChorusOS Applications

The ChorusOS operating system provides an environment for applications running on a network of target machines, controlled by a remote host.

- The target system runs the ChorusOS operating system and provides the execution environment.
- The host machine provides the development and debugging environment. The user can develop the applications on the host and, from the host, start and debug these applications on the targets.

Programming Conventions

Services provided by the ChorusOS operating system are accessed as C routines. C header files provide the required constants, types and prototypes definitions. As the ChorusOS operating system is highly modular, header files reflect this modularity. However, in the following examples a global header file, named `chorus.h`, which collects most of the required header files, has been used for simplicity. Please refer to the man pages to get the actual minimum header file required for each service.

Most ChorusOS operating system constants start with `K_`. ChorusOS operating system error codes start with `K_E`. Constants and error codes are all written in uppercase.

Most specific data types are prefixed by `Kn`. When type names are composed of several lexems, the first letter of each lexem is written in uppercase while other letters are in lowercase, as in `KnRgnDesc` (region descriptor).

General Principles

In order to compile and link an application, the following information is needed:

- The header files and compilation flags
- The program entry point
- The libraries to be linked with the program according to the services used by the application, the environment present on the target system, and the actor type (user or supervisor)

Program Entry Point

In order to initialize the libraries correctly before starting the execution of the application code, the program entry point must be set to `_start`. After the initialization of libraries is completed, `_start` calls the `_main` routine which initializes variables in C++ programs. The `main()` routine is then called.

The `_main` routine manages any double calling at program initialization; some C++ compilers force a call to `_main` at the beginning of `main()`.

Depending on the development system, it may be necessary to use specific linker directives to force the linker to extract the `_start` and `_main` routines from the libraries.

Libraries

In order to choose which ChorusOS operating system libraries to use, the following points need to be considered:

- Which APIs are used by the application program. For example, a program using the mathematical API has to be linked with the `libm.a` library.
- What type of system is running on the target. For example, the `librpc.a` library cannot be used if you are using the basic environment and no additional features.
- The address space in which the program will execute. For example a program loaded as a user extended actor must be linked with the `libcx.a` library.

Supervisor Actor Binaries

As supervisor actors share the same supervisor address space, they are built as relocatable binaries, leaving the choice of the final link addresses to either the system configuration utility building the system image (for the basic environment) or the Actor Manager (for the extended environment).

Care must be taken when programming supervisor actors: no memory protection is provided between supervisor actors. A badly written supervisor actor accessing addresses outside its own address space can corrupt any supervisor region and cause unexpected behavior such as a system crash or reboot.

User Actor Binaries

User actors are also built as relocatable binaries, even though they use private address spaces. The link address of the user actors and the size of the user address space are board dependent. For a given board, all user actors are linked at the same address.

The final link is done by the Actor Manager when actors are loaded dynamically on the target.

Application Programming Interfaces

This section provides an overview of all programming interfaces available for applications developed for the ChorusOS operating system. The programming interface may differ from one program to another depending on:

- Its execution environment: basic or extended environment.
- Its execution mode: running in user or supervisor space.
- Its execution structure: containing one or more ChorusOS operating system threads.

Naming Conventions

Library names in the ChorusOS operating system use the following conventions with regard to their suffixes:

.u.a	These libraries can only be used to build actors that will be loaded in a user address space.
.s.a	These libraries can only be used to build actors that will be loaded in the supervisor address space.
.a	These libraries can be used to build any type of actor.

Note - When a library has both a user and supervisor version, it will be referred to using the .a suffix only.

All header file and library pathnames listed in the next subsections are related to the installation path of your ChorusOS delivery, typically
`/opt/SUNWconn/SEW/4.0/chorus-<target>`.

Basic Environment APIs

The programming environment of basic actors consists of the following interfaces:

- The Microkernel API
- The Private Data API
- The Standard-C API
- The Console Input/Output API

All routines implementing these APIs have been grouped into two libraries:

<code>kernel/lib/embedded/libebd.u.a</code>	for user actors
<code>kernel/lib/embedded/libebd.s.a</code>	for supervisor actors

ChorusOS actors using the Basic Environment API are called *embedded* actors.

Extended Environment API

The programming environment of extended actors consists of the following interfaces:

- The Microkernel API
- The Private Data API
- The Standard-C API
- The POSIX Input/Output API
- The POSIX Network API
- The Actor Management API

All routines implementing these APIs have been grouped into one library:

<code>os/lib/classix/libcx.a</code>	for user and supervisor actors
-------------------------------------	--------------------------------

Note - An extended supervisor actor should not use the `svExceptionHandler()` call as an extended actor inherits the Actor Manager exception handler.

Other APIs

Other APIs are provided with the ChorusOS operating system. Depending on their nature, they may be available to both basic and extended environments or restricted to a single environment. The following subsections give a description of the libraries implementing these APIs.

POSIX Micro Real-time Profile API

Routines implementing the MRTP (Micro Real-time Profile) API are included within the `libcx` and `libebd` libraries. They are available to both basic and extended actors.

Mathematical API

Routines implementing the Mathematical API are packaged in an independent library `kernel/lib/libm/libm.a`. This library is available to both basic and extended actors.

Sun RPC API

Routines implementing the Sun RPC API are packaged in an independent library `os/lib/classix/librpc.a` which is not thread-safe. This API is restricted to extended actors.

GNU 2.7.1 C++ API

The C++ library `os/lib/CC/libC.a` provides support for C++ applications with a complete and thread-safe library package. Every service offered by `libC.a` ensures that shared data is only accessed after signaling the relevant synchronization objects.

To allow atomic manipulation of any stream class (`iostream` or `fstream` for example), the API of `libC.a` has been extended with the following two services:

```
ios::lock( )  
  
ios::unlock( )
```

The `ios::lock()` service is used to lock any stream class object. The `ios::unlock()` service is used to unlock any stream class object. All services called upon a given stream object `StrObj` which are preceded by `StrObj.lock()` and followed by `StrObj.unlock()` are executed in an atomic way. It is guaranteed that no other thread can access `StrObj` as long as the lock is on.

An I/O stream object can be locked in two ways. For example, if `cout` is an I/O stream object:

```
cout.lock();
cout << "atomic " << "output";
... (any other operation on cout)
cout.unlock();
```

In this case the member function `ios::lock()` is called.

The following syntax could also be used:

```
cout << lock << "atomic " << "output" << unlock;
```

Embedded C++ actors can be linked with `os/lib/CC/libC.a` if they do not make use of the `iostream` and `fstream` packages.

Multithreading

The `libebd.a`, `libcx.a`, `libm.a` and `libC.a` libraries have been made thread-safe in order to support multithreaded actors. This is managed by the library in the following way:

- by protecting shared variables with mutexes and using `threadOnce()` to initialize these mutexes
- by using the Private Data library to maintain private variables per thread (for `errno` management, see the next section)

Defining `errno` as one global variable for the actor is not suitable for multithreaded actors as situations can arise where a thread, examining `errno` on return from a failed system call, concludes that the call failed for the wrong reason because the global `errno` was changed by another system call in another thread in the meantime. Some programs also test `errno` rather than system call return values to detect errors.

To avoid this, the header file `errno.h`, exported by the extended environment, should be included in any source file using `errno`. This will result in a separate value for `errno` for each thread.

Header Files

The ChorusOS operating system header files are packaged in five different directories:

- `kernel/include/chorus`. Header files in this directory export the following APIs:
 - The Microkernel API
 - The Private Data API
 - The Actor Management API
- `kernel/include/stdc`. Header files in this directory export the following APIs:
 - The Standard-C API
 - The Mathematical API
 - The Console Input/Output API
 - Some BSD specific APIs
- `include/posix`. Header files in this directory export the following APIs:
 - The Standard-C API
 - The Mathematical API
 - The POSIX Input/Output API
 - The POSIX MRTP API
 - The POSIX Network API
 - The Sun RPC API
 - Some BSD specific APIs
- `include/CC`. Header files in this directory export the GNU 2.7.1 C++ API.
- `include/X11`. Header files in this directory export the following libraries:
`libX11.a, libXaw.a, libXext.a, libXmu.a, libXt.a.`

Typical ChorusOS operating system applications use header files of the `include/chorus` and `include/posix` directories (and also `include/CC` for applications using the GNU 2.7.1 C++ API).

Developing personality servers (such as servers implementing a UNIX personality) on a ChorusOS operating system needs extra care in order to avoid conflicts between data types declared by ChorusOS operating system header files, and data types declared by the server's header files. These servers should be restricted to header files of the `include/chorus` and `include/stdc` directories and use the `_CHO_POSIX_TYPES_NOTDEF` compile option.

Developing ChorusOS Applications

This section explains how to build a component to be included in a ChorusOS operating system. The component could be an application, a device driver, or a BSP. To build a ChorusOS component, you use the `make` and `imake` tools. All development tools are provided in the `tools` directory of your delivery.

make Environment

The `make` environment is defined by a file containing variable definitions and rules. Rules for compiling C, C++, and assembly language are provided. The rules are specific to the compiler you use, and the name of the file indicates the compiler. For example, if you are using the `gcc` compiler, the `make` environment file is called `tgt-make/gcc-devsys.mk`. The file contains the variables and rules required for building the component. The following variables are defined:

- `CFLAGS` and `CXXFLAGS` specify the compilation options for C and C++ files, respectively. The compilation options are shown in Table 4-1.

TABLE 4-1 Compilation Options

Option	Possible Settings	Default Setting
<code>WARN</code>	<code>WARN_ON</code> , <code>WARN_OFF</code>	<code>WARN_ON</code>
<code>DEBUG</code>	<code>DEBUG_ON</code> , <code>DEBUG_OFF</code>	<code>DEBUG_OFF</code>
<code>PROF</code>	<code>PROF_ON</code> , <code>PROF_OFF</code>	<code>PROF_OFF</code>
<code>OPT</code>	<code>OPT_ON</code> , <code>OPT_OFF</code>	<code>OPT_ON</code>

- `INCLUDES` and `DEPENDS` specify include and depend values. These variables can be overloaded at the application level. They are grouped into the `CPPFLAGS` flag, which is used in compilation and to compute dependencies. Both `INCLUDES` and `DEPENDS` can be initialized at the application level.
- `LD_UCRT0`, `LD_SCRT0`, `LD_LCRT0`, `LD_CRTI`, `LD_CRTN`, and `LD_CRTXT` are used to manage different types of `crt` object files.
- `LD_U_ACTOR` and `LD_S_ACTOR` specify link information for user and supervisor actors.
- `CLX_U_LIBS`, `CLX_S_LIBS`, `EBD_U_LIBS`, `EBD_S_LIBS`, and `CXX_LIBS` are provided to manage libraries.

The `make` environment includes the following commands: `cc`, `ld`, `as`, and `mkactors`.

imake Environment

The ChorusOS `imake` environment extends the `make` environment by providing template rules for common ChorusOS build operations through generic names. When using those predefined `imake` rules, you do not need to know which libraries, `crt` files, or entry points you should use to build an application, as they are automatically selected for you.

Instead of creating Makefiles you must create Imakefiles, as `imake` will generate Makefiles from them.

The `imake` environment is defined by four files containing sets of variables and rules, located in the `tools/imake` directory. The rules are independent of the compiler you use.

- `Imake.tmpl` contains definitions of variables.
- `Imake.rules` contains build rules. See “`imake` Build Rules” on page 104.
- `Package.rules` contains the rules used to build a binary distribution.

imake Variable Definitions

The file `Imake.tmpl` contains the following definitions:

- `FAMILY`, indicating the target family (`x86`, `usparc`, `ppc60x`, `mpc860`, `mpc8260`).
- `COMPILER`, indicating the compiler to be used (`gcc`, for example).
- `REL_DIR`, indicating the path of the current directory. This variable is automatically set in subdirectories by `imake`.
- `HOSTOS`, indicating the host operating system (`solaris`, `win32`).
- `DEVTOOLS_DIR`, indicating the path of the ChorusOS tools.

imake Build Rules

The file `Imake.rules` contains macros known as Imake build rules. Their name and function are described in Table 4-2.

TABLE 4-2 Imake build rules

Macro name	Function
<code>MakeDir(dir)</code>	Creates the directory named <code>dir</code> .
<code>LibraryTarget(lib, objs)</code>	Adds the objects indicated by <code>objs</code> into the library <code>lib</code> .
<code>Depend(srcs)</code>	Computes the dependencies of <code>srcs</code> and adds them to the dependency list in the Makefile (using <code>makedepend</code>).
<code>ActorTarget(prog, objs, options, crt0, libs)</code>	Uses <code>objs</code> to create a C actor called <code>prog</code> , and passes <code>options</code> , <code>crt0</code> and <code>libs</code> to the linker.
<code>UserActorTarget(prog, objs, libs)</code>	Creates a user C actor.
<code>SupActorTarget(prog, objs, libs)</code>	Creates a supervisor C actor.
<code>EmbeddedUserActorTarget(prog, objs, libs)</code>	Creates an embedded user C actor.
<code>EmbeddedSupActorTarget(prog, objs, libs)</code>	Creates an embedded supervisor C actor.
<code>BuiltinDriver(prog, objs, libs)</code>	Creates a ChorusOS operating system driver.
<code>BspProgtarget(prog, entry, objs, libs)</code>	Creates a BSP program.
<code>CXXActorTarget(prog, objs, options, crt0, libs)</code>	Uses <code>objs</code> to create a C++ actor called <code>prog</code> , and passes <code>options</code> , <code>crt0</code> and <code>libs</code> to the linker.
<code>CXXUserActorTarget(prog, objs, libs)</code>	Creates a user C++ actor.
<code>CXXSupActorTarget(prog, objs, libs)</code>	Creates a supervisor C++ actor.
<code>CXXEmbeddedUserActorTarget(prog, objs, libs)</code>	Creates an embedded user C++ actor.
<code>CXXEmbeddedSupActorTarget(prog, objs, libs)</code>	Creates an embedded supervisor C++ actor.
<code>DynamicUserTarget(prog, objs, libs, dynamicLibs, dlDeps, options)</code>	Creates a dynamic user C actor.
<code>DynamicSupTarget(prog, objs, libs, dynamicLibs, dlDeps, options)</code>	Creates a dynamic user C actor.

TABLE 4-2 Imake build rules *(continued)*

Macro name	Function
<code>DynamicCXXUserTarget(prog, objs, libs, dynamicLibs, dlDeps, options)</code>	Creates a dynamic user C actor.
<code>DynamicCXXSupTarget(prog, objs, libs, dynamicLibs, dlDeps, options)</code>	Creates a dynamic user C actor.
<code>DynamicLibraryTarget(dlib, objs, staticLibs, dynamicLibs, dlDeps, options)</code>	Creates a dynamic library.

Rules used to build actors use the following common arguments:

- `obj` is the list of binary objects included in the actor.
- `prog` is the name of the actor.
- `libs` is the list of additional libraries used to build the actor. The actor is linked by default with the library to provide either the basic or extended environment.

The rules used to build dynamic actors are described in more detail in “Building a Dynamic Program” on page 112.

imake Packaging Rules

The file `Package.rules` contains macros known as Imake packaging rules for building a binary distribution. Their name and function are described in Table 4-3.

TABLE 4-3 Imake packaging rules

Macro name	Function
<code>DistLibrary(lib, dir)</code>	Creates the directory <code>dir</code> and copies the library <code>lib</code> into it.
<code>DistActor(actor, dir)</code>	Creates the directory <code>dir</code> and copies the actor <code>actor</code> into it.
<code>DistFile(file, dir)</code>	Creates the directory <code>dir</code> and copies the file <code>file</code> into it.
<code>DistRenFile(file, nFile, dir)</code>	Creates the directory <code>dir</code> , copies <code>file</code> into it, changing the name of <code>file</code> to <code>nFile</code> .
<code>DistProgram(program, dir)</code>	Creates the directory <code>dir</code> and copies <code>program</code> into it.

Examples

Simple imake Example

The application in this example is composed of a single C source file, for example `myprog.c` in the directory `myprog`. Writing an `Imakefile` is quite straightforward. First, you must set the `SRCS` variable to the list of source files (in this case only one).

```
SRCS = myprog.c
```

Then, you must specify how to build the executable. The macro you use depends on the type of binary you want. If you want to build a user-mode binary (for example `myprog_u`), use the `UserActorTarget()` macro, as illustrated below. The first argument is the name of the executable. The second argument lists the object files. The third argument allows you to specify which libraries your program depends on. In this example there is no library, hence the empty argument (you could also pass `NullParameter`).

```
UserActorTarget(myprog_u,myprog.o,)
```

If you want to build a supervisor-mode binary (for example, `myprog_s.r`), use the `SupActorTarget()` as shown below. The arguments are the same as for `UserActorTarget()`.

```
SupActorTarget(myprog_s.r,myprog.o,)
```

Finally, use the `Depend()` macro to generate the `Makefile` dependencies.

```
Depend($(SRCS))
```

The `Imakefile` is complete. It looks like this:

```
SRCS = myprog.c
UserActorTarget(myprog_u,myprog.o,)
SupActorTarget(myprog_s.r,myprog.o,)
Depend($(SRCS))
```

Next, generate the Makefile with the ChorusOSMkMf tool (see the ChorusOSMkMf(1CC) manpage for details). In the `myprog` directory, type:

```
% ChorusOSMkMf build_dir
```

Where `build_dir` is the directory where you have built a ChorusOS system image on which your application will run.

Next, generate the make dependencies by typing the following command:

```
% make depend
```

Finally, compile and link the program by typing:

```
% make
```

The program is now ready to be executed, and can be run on your target by following the steps in “Running the “Hello World” Example” on page 65.

imake with Multiple Source Files

If an application used source files located in several subdirectories, you need to create a root Imakefile in the root directory, containing only the following:

```
#define IHaveSubdirs  
SUBDIRS = subdir1 subdir2 ...
```

where `subdir1`, `subdir2`, ... are the subdirectories containing the source files (or other intermediate root Imakefile files). Next, create an Imakefile in each subdirectory containing source files. To generate the first Makefile, go to the root directory and type:

```
% ChorusOSMkMf build_dir
```

Next, populate the tree with Makefile files, generate dependencies and finally compile the programs by typing the `make Makefiles`, `make depend`, and then `make` commands.

```
% make Makefiles
% make depend
% make
```

The program is now ready to be executed.

Note - Examples of Imakefiles which you can modify and use to build your own applications are provided in

`/opt/SUNWconn/SEW/4.0/chorus-<target>/src/opt/examples.`

Using Dynamic Libraries

There are two types of library in the ChorusOS operating system:

- *Static*

Static library names are suffixed by `.a`. A static library is a collection of binary object files (`.o`). The linker concatenates all needed binary objects of the static libraries into the executable program file.

- *Dynamic*

Dynamic library names are suffixed by `.so`. They can be linked with a program at runtime. Dynamic linking is supported by a ChorusOS operating system component called the runtime linker. It occurs in two cases:

- At actor start-up: in order to build the executable, the runtime linker loads and links a list of libraries. These libraries are called the dependencies of the executable.
- During actor execution: with the dynamic linking API, an actor can explicitly load and link dynamic libraries, using the `dlopen()` function. This allows dynamic programming.

Dynamic libraries are loaded at runtime and are not included in the executable. This reduces the size of executable files. Dynamic libraries use relocatable code format. This code is turned into absolute code by the runtime linker.

In the ChorusOS operating system, both user and supervisor actors (but not boot actors) can use dynamic libraries.

Relocatable code can be contained in two types of executable:

- *Relocatable executable*: at actor start-up, the runtime linker loads the executable and performs the necessary relocations.
- *Dynamic executable*: at actor start-up, the runtime linker loads the executable and performs the necessary relocations. It also loads and links the executable dependencies, that is, the dynamic libraries used by the executable.

An actor that uses dynamic libraries is called a *dynamic actor*. A *relocatable actor* uses only static libraries.

Static and Dynamic Linking

The following table summarizes the actions performed by the static linker, which runs on the development host, and by the runtime linker, which runs on the target.

Link	Relocatable Executable	Dynamic executable
Static Linker	.a Static linker adds necessary objects (.o) of a static library (.a) to the executable.	.a Static linker adds necessary objects (.o) of a static library (.a) to the executable. .so Static linker adds the library to the list of libraries to load at actor start-up (afexec).
Runtime Linker (afexec)	-	.so At actor start-up, libraries are loaded and linked by the runtime linker. Libraries to load are defined either at static link, or in the LD_PRELOAD environment variable. The runtime linker uses a library search path to find dynamic and shared libraries.
Runtime Linker (dlopen)	-	.so Application explicitly asks the runtime linker to dynamically load and link a dynamic library, using the dlopen() function of the dynamic linking API.

Dynamic linking of libraries applies recursively to library dependencies: when a library is loaded, all the libraries it uses are also loaded.

Building a Dynamic Library

This section describes how to build dynamic and shared programs using the `imake` tool. See “Developing ChorusOS Applications” on page 103 for more general information about using `imake`.

The following `imake` macro builds dynamic libraries:

```
DynamicLibraryTarget(dlib, objs, staticLibs, dynamicLibs, dlDeps, options)
```

- `dlib`: name of resulting dynamic library (must be suffixed by `.so`).
- `objs`: library components: list of binary object files (suffixed by `.o`).
- `staticLibs`: list of static libraries (`.a`) that will be statically linked.
- `dynamicLibs`: list of dependencies: dynamic libraries that must be loaded together with the resulting library. Each library can be defined in one of two ways:
 1. `-L path -l name`: on the host, the linker will look for library `path/libname.so`. On the target, the runtime linker will look for `libname.so` in the library search path.
 2. `path`: this is an absolute or relative library path used on the host by the linker, and on the target by the runtime linker. A relative path containing a `/` is interpreted as relative to the current directory by the runtime linker. A path without `/` is searched in the library search path by the runtime linker.
- `dlDeps`: list of dynamic libraries the library depends upon. If these libraries are changed, the resulting library `dlib` will be rebuilt. Each library must be defined as a path on the host. Generally `dlDeps` duplicates the libraries described in `dynamicLibs`. This allows, when the `-L path -l name` syntax is used, to express the dependency without embedding a path in the executable.
- `options`: any linker options, preceded by `-Xlinker`. This must be used to supply system-specific linker options which GNU C does not know how to recognize. If you want to pass an option that takes an argument, you must use `-Xlinker` twice, once for the option and once for the argument.

The following example builds a dynamic library named `libfoo.so` from the binary objects files `a.o` and `b.o`. When this library is loaded dynamically, the runtime linker will also load the dynamic library `libdyn.so`, which must be in its search path.

```
DynamicLibraryTarget(  
    libfoo.so,  
    a.o b.o, ,  
    libdyn.so, , )
```

Building a Dynamic Program

The following imake macros build dynamic executables:

```
DynamicUserTarget(prog, objs, staticLibs,  
                  dynamicLibs, dlDeps, options)  
DynamicSupTarget(prog, objs, staticLibs,  
                  dynamicLibs, dlDeps, options)  
DynamicCXXUserTarget(prog, objs, staticLibs,  
                      dynamicLibs, dlDeps, options)  
DynamicCXXSupTarget(prog, objs, staticLibs,  
                     dynamicLibs, dlDeps, options)  
DynamicLibraryTarget(prog, objs, staticLibs,  
                      dynamicLibs, dlDeps, options)
```

The `prog` argument is the name of the resulting program. Other arguments are the same as the `DynamicLibraryTarget()` macro. For the `options` argument, the following options are particularly useful:

- `-Xlinker -soname=<name>` within a `DynamicLibraryTarget()` rule sets the internal so-name of the library. If a library used as a dependency in a rule that builds a dynamic executable has a so-name defined, the executable records the so-name instead of the `dynamicLibs` argument.
- `-Xlinker -rpath -Xlinker <dir>`: defines the runpath directory that is added to the library search path (see “Runtime Linker” on page 113).

The following example builds a dynamic program named `prog` from the binary object files `a.o` and `b.o`. The program is statically linked with the static ChorusOS operating system library. When this program is started, the runtime linker will load the dynamic library `libdyn.so`. In the target file system, this library can be located in the `/libraries` directory, as this directory is added to the search path of the runtime linker.

```
DynamicUserTarget(  
    prog,  
    a.o b.o ,  
    libdyn.so ,  
    -Xlinker -rpath -Xlinker /libraries)
```

Dynamic Programming

The previous sections described how to use dynamic objects (dynamic and shared libraries) that are loaded by the runtime linker at actor start-up. In addition to this mechanism, an actor can also bind a dynamic or shared library explicitly during its execution. This on-demand object binding has several advantages:

- By processing a dynamic object when it is required rather than during the initialization of an application, start-up time can be greatly reduced. In fact, the object might not be required if its services are not needed during a particular run of the application.
- The application can choose between several different dynamic objects depending on the exact services required. For example, if different libraries implement the same driver interface, the application can choose one driver implementation and load it dynamically.
- Any dynamic object added to the actor address space during execution can be freed after use, thus reducing the overall memory consumption.

Typically, an application performs the following sequence to access an additional dynamic object, using the dynamic library API:

- A dynamic object is located and added to the address space of a running application using `dlopen()`. Any dependencies this dynamic object has are located and added at this time.
- The added dynamic object and its dependencies are relocated, and any initialization sections within these objects are called.
- The application locates symbols within the added objects using `dlsym()`. The application can then reference the data or call the functions defined by these new symbols.
- After the application has finished with the objects, the address space can be freed using `dlclose()`. Any termination section within the objects being freed will be called at this time.
- Any error conditions that occur as a result of using these runtime linker interface routines can be displayed using `dlerror()`.

Runtime Linker

This section describes the functions performed by the runtime linker, as well as the features it supports for dynamic applications.

Dynamic applications consist of one or more dynamic objects. They are typically a dynamic executable and its dynamic object dependencies. As part of the initialization of a dynamic application, the runtime linker completes the binding of the application to its dynamic object dependencies.

In addition to initializing an application, the runtime linker provides services that allow the application to extend its address space by mapping additional dynamic objects and binding to symbols within them.

The runtime linker performs the following functions:

- It analyzes the executable's dynamic information section and determines which dynamic libraries are required.
- It locates and loads these dynamic libraries, and then it analyzes their dynamic information sections to determine whether any additional dynamic library dependencies are required.
- Once all dynamic libraries are located and loaded, the runtime linker performs any necessary relocations to bind these dynamic libraries in preparation for actor execution.
- It calls any initialization functions provided by the dynamic libraries. These are called in the reverse order of the topologically sorted dependencies. Should cyclical dependencies exist, the initialization functions are called using the sorted order with the cycle removed.
- It passes control to the application.
- It calls any finalization functions on deletion of dynamic objects from the actor. These are called in the order of the topologically sorted dependencies.
- The application can also call upon the runtime linker's services to acquire additional dynamic objects with `dlopen()` and bind to symbols within these objects with `dlsym()`.

The runtime linker uses a prescribed search path for locating the dynamic dependencies of an object. The default search paths are the runpath recorded in the object, followed by `/usr/lib`. The runpath is specified when the dynamic object is constructed using the `-rpath` option of the linker. The environment variable `LD_LIBRARY_PATH` can be used to indicate directories to be searched before the default directories.

Note - The runtime linker needs a file system to load dynamic objects. This file system can be on a host and accessed through NFS from the target. In embedded systems without a network connection, a bank of the system image can be used. For example: `/image/sys_bank`.

Environment Variables

The following environment variables are used by the runtime linker:

- `LD_LIBRARY_PATH` specifies a colon (:) separated list of directories that are to be searched before the default directories defined above. This is used to enhance the search path that the runtime linker uses to find dynamic and shared libraries.
- `LD_PRELOAD` provides a dynamic object name that is linked after the program is loaded but before any other dynamic objects that the program references.
- `LD_DEBUG` is a column-separated list of tokens for debugging the runtime linking of an application. Each token is associated with a set of traces which are

displayed on the system console during runtime linking. The supported tokens are: `file-ops`, `reloc`, `symbol-resolution`, `malloc`, `segment-alloc`, `dependancy`, `misc`, `linking`, `dynamic-map-op`, `group`, and `error`. Wildcard substitutions are also allowed, so that `s*`, for example, matches both `symbol-resolution` and `segment-alloc`, and `*` matches all traces.

Supported Features

- Immediate binding:

The runtime linker performs both data reference and function reference relocations during process initialization, before transferring control to the application. This behavior is equivalent to the `LD_BIND_NOW` behavior in the Solaris operating environment (lazy binding is not supported).

- No version checking:

The runtime linker performs no version dependency checking. When looking for a library, the runtime linker looks for a file name matching the library name exactly. This behavior is equivalent to the `LD_NOVERSION` behavior in the Solaris operating environment.

- Weak symbols and aliases:

During symbol resolution, weak symbols will be silently overridden by any global definition with the same name. Weak symbols can be defined alone or as aliases to global symbols. Weak symbols are defined with pragma definitions.

Examples

This section contains two examples of dynamic programs. In all these examples, it is assumed that a standard development environment has been set up: system build tree, search path, boot and initialization of target machine. It also assumes that the *chorus_root_directory* is the path of the target root directory on the NFS host (for example `/home/chorus/root`), the name of the target is `jericho`, and the environment variable `WORK` refers to the directory used for building these examples

Dynamic Link at Actor Start-up

The following dynamic program uses a custom dynamic library which will be loaded and linked at actor start-up. It uses a function `foo()` which is defined in the dynamic library. This function calls a function `bar()` defined in the main program.

This is the dynamic program `progdyn.c`:

```
#include <chorus.h>

extern void foo();
```

```

main() {
    foo();                /* calling foo defined in the library */
}

void bar() {
    printf ("bar called\n");
}

```

This is the dynamic library libdyn.c:

```

#include <chorus.h>

extern void bar();

void foo() {
    printf ("Calling bar\n");
    bar();                /* calling bar defined in the main program */
}

```

Building the Dynamic Library

Create a directory libdyndir in \$WORK, containing libdyn.c and the following Imakefile:

```

SRCS = libdyn.c
DynamicLibraryTarget (libdyn.so, libdyn.o, , , , -Xlinker -soname=libdyn.so )
Depend(libdyn.c)

```

In the libdyndir directory, build the dynamic library libdyn.so using the ChorusOSMkMf, make depend, and make commands.

Building the Dynamic Program

Create a directory progdyndir in \$WORK, containing progdyn.c and the following Imakefile:

```

SRCS = progdyn.c
DynamicUserTarget (progdyn, progdyn.o, ,
    $(WORK)/libdyndir/libdyn.so,
    $(WORK)/libdyndir/libdyn.so, )
Depend()

```

In the progdyndir directory, build the dynamic program progdyn using the ChorusOSMkMf, make depend and make commands.

```
% ChorusOSMkMf $WORK
% make depend
% make
```

Running the Dynamic Program

Copy the dynamic program into the `/bin` subdirectory of the *chorus_root_directory* directory:

```
% cp $WORK/progdyndir/progdyn chorus_root_directory/bin
```

Copy the dynamic library into the `/lib` subdirectory of the *chorus_root_directory* directory:

```
% cp $WORK/libdyndir/libdyn.so chorus_root_directory/lib
```

Then, the following command will tell the runtime linker where to find the `libdyn.so` dynamic library:

```
% rsh jericho setenv LD_LIBRARY_PATH /lib
```

Alternatively, set the `runpath` to `/lib` in the `ldopts` argument of the program macro (`-rpath /lib`).

Finally, the following command will start the program and dynamically load the `libdyn.so` library:

```
% rsh jericho arun /bin/progdyn
```

Explicit Link Using `dlopen`

The following program explicitly loads a dynamic library at runtime, using the function `dlopen()`. It searches for the address of the `dynfunc()` function defined in the library and calls this function.

This is the dynamic program `progdyn2.c`:

```

#include <chorus.h>
#include <cx/dlfcn.h>

int main()
{
    void    (*funcptr)();          /* pointer to function to search */
    void    *handle;              /* handle to the dynamic library */

    /* finding the library */
    handle = dlopen ("libdyn2.so", RTLD_NOW);
    if !(handle) { printf ("Cannot find library libdyn2.so\n"); exit(1); }

    /* finding the function in the library */
    funcptr = (void (*)()) dlsym (handle, "dynfunc");
    if !(funcptr) { printf ("Cannot find function dynfunc\n"); exit(1); }

    /* calling library function */
    (*funcptr)();
}

```

This is the dynamic library libdyn2.c:

```

#include <chorus.h>

void dynfunc() {
    printf ("Calling dynfunc\n");
}

```

Building the Program and the Library

The program and library above are built in the same way as in the previous example, using two Imakefiles:

Create a directory libdyn2dir in \$WORK, containing libdyn2.c and the following Imakefile:

```

SRCS = libdyn2.c
DynamicLibraryTarget (libdyn2.so, libdyn2.o, , , , )
DependTarget(libdyn2.c)

```

Create a directory progdyn2dir in \$WORK, containing progdyn2.c and the following Imakefile:

```

SRCS = progdyn2.c
DynamicUserTarget (progdyn2, progdyn2.o, , , , )
Depend(progdyn2.c)

```

Running the Dynamic Program

Copy the dynamic program into the `/bin` subdirectory of the *chorus_root_directory* directory:

```
% cp $WORK/progdyn2dir/progdyn2 chorus_root_directory/bin
```

Copy the dynamic library into the `/lib` subdirectory of the *chorus_root_directory* directory:

```
% cp $WORK/libdyn2dir/libdyn2.so chorus_root_directory/lib
```

Then, the following command will tell the runtime linker where to find the `libdyn2.so` dynamic library:

```
% rsh jericho setenv LD_LIBRARY_PATH /lib
```

Finally, the following command will start the program:

```
% rsh jericho arun /bin/progdyn2
```

At program start-up, the runtime linker will only load the executable `progdyn2`. The `libdyn2.so` library will be loaded when the `dlopen()` function is called.

Using Actors

This chapter explains the role of actors in a ChorusOS operating system application. It contains the following sections:

- “Actor Definition” on page 121 defines the term *actor*, and explains how an actor is named and used.
- “Loading Actors” on page 126 explains how to load an actor.
- “Execution Environment of Actors” on page 127 explains how the execution environment of an actor is defined.
- “Spawning an Actor” on page 130 explain the ways in which an actor can be run.

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. They will be covered in more detail throughout this chapter. 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. They 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 target machine where the ChorusOS operating system is running. When properly configured, 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 which is automatically created at load time by the system. The ChorusOS operating system provides means to dynamically create and run more than one thread in an actor. It also offers services which enable these actors, whether single-threaded or multithreaded, to cooperate, synchronize, exchange data either locally or remotely, or get control of hardware events, for example. These topics will be covered step by step throughout this chapter.

An actor may be of two types: it may be either a supervisor actor or a user actor. This information defines 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.

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 may also be of two kinds: 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 which may not be changed. A relocatable file is a binary which may 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 more detail in "User and Supervisor Actors" on page 123.

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:

K_MYACTOR.

In addition, an actor created from the POSIX personality is identified by a local actor identifier. This actor identifier is displayed on the console as the result of the `arun` command. It may be used from the console as a parameter of the `akill` command.

```
% rsh target arun hello
Started aid = 13
%
```

In this example, *target* is the name of your target.

User and Supervisor Actors

There are two main kinds of actors which may be 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 which is not valid in its address space, it will encounter a fault and, by default, will be 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 5-1). 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 the actor, the user does not need to be aware of the partitioning of the supervisor address space: using relocatable binary files will suffice.

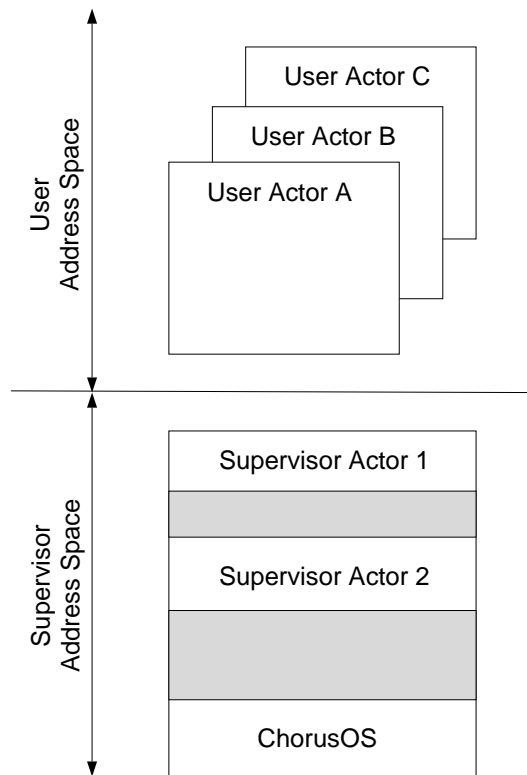


Figure 5-1 User and Supervisor Address Spaces

The ChorusOS operating system offers a way to determine dynamically whether a program is currently running as a user or a supervisor actor:

```
#include <chorus.h>

int actorPrivilege(KnCap* actorCap,
                  KnActorPrivilege* old,
                  KnActorPrivilege* new);
```

If `actorCap` is set to the name of an actor, you can use this API to obtain the privilege of the named actor. If `actorCap` is set to the predefined value `K_MYACTOR`, you can obtain the privilege of the current actor. This call may also be used to dynamically change the privilege of an actor from user to supervisor or vice versa.

The following example illustrates a usage of the `actorPrivilege()` service. It is a small program that retrieves its privilege, without trying to modify it. It prints one message if the actor is running as a user actor, and another if it is running as a supervisor actor.

CODE EXAMPLE 5-1 Getting Actor Privilege

```
#include <stdio.h>
#include <chorus.h>

int main(int argc, char** argv, char** envp)
{
    KnActorPrivilege    actorP;
    int                 res;

    /* Get actor's privilege */
    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }

    if (actorP == K_SUPACTOR) {
        printf("This actor is running as a supervisor actor\n");
    } else {
        printf("This actor is running as a user actor\n");
    }

    exit(0);
}
```

`KnActorPrivilege` is the type defined by the ChorusOS operating system to handle the type of an actor. The defined values for the actor type are:

- `K_SUPACTOR`: a supervisor actor running in the supervisor address space which can access all privileged kernel calls.
- `K_SYSTEMACTOR`: a trusted user actor, launched at boot time by the kernel and running in its own user space address, which can access certain privileged kernel calls.
- `K_USERACTOR`: a user actor running in its user space. It has fewer privileges than `K_SYSTEMACTOR`.

Loading Actors

Actors may be loaded in two different ways: either at system boot time or dynamically.

The ChorusOS operating system is started from a bootable file, called the system image, which is loaded in memory either by a hardware boot or a primary boot, depending on the hardware. This bootable file contains the image of the system to be run on the target machine.

Boot Actors

The ChorusOS operating system environment provides tools to configure this system image with user provided actors, which may be user or supervisor actors. Once the system has performed its own initialization, it starts these actors automatically, creating a main thread in each of them. These actors are often referred to as boot actors.

Loading Actors Dynamically

In order to be able to dynamically load an application on a ChorusOS operating system, the system must have been configured with the `ACTOR_EXTENDED_MNGT` feature. In this type of configuration, the ChorusOS operating system is able to dynamically load binary files from the host system acting as an NFS server, from a local disk, or from the system image (`/image/sys_bank`). This host-target environment enables the user to load supervisor and user actors using a simple remote shell mechanism. To execute an application called `hello` on the target host moon, use the `arun` command, as follows:

```
% rsh moon arun hello
```

The ChorusOS operating system uses the `.r` suffix to denote relocatable binary files.

A relocatable actor is executed as follows:

```
% rsh moon arun mySupAppl.r
```

In this example, the `.r` suffix could be omitted, because the ChorusOS operating system looks first for the name as specified, `mySupAppl`, and then, if it does not find

a file of that name, automatically looks for a file of that name with the suffix `.r`, `mySupAppl.r`.

Execution Environment of Actors

The execution environment of actors varies slightly depending on whether the actors have been loaded dynamically or at boot time.

An actor loaded at boot time does not have any arguments or environment. If it is linked with the embedded library, it may perform very simple input or output operations such as printing traces on the system console using the `printf()` C library routine. It may also read characters typed in from the keyboard of the system console through the `scanf()` C library routine. If it is linked with the `libcx.a` library, it must first open `/dev/console` three times in order to activate `stdin`, `stdout`, or `stderr` and allow `printf()` or `scanf()` operations on the system console. The main thread of an actor loaded at boot time will belong to the `SCHED_FIFO` scheduling policy, with an arbitrary priority depending on the rank of the actor within the system image. The size of the stack provided to the main thread of this type of actor is defined by a system-wide tunable parameter.

A dynamically loaded actor is started as a regular C program with arguments and environments:

```
int main(int argc, char** argv, char** envp)
{
    /* Main routine of a dynamically loaded actor */
    /* regardless of whether the actor is a user */
    /* or supervisor actor. */
}
```

The standard input, output and error files of an extended actor may be redirected so that the I/O operations performed by this actor occur either on the system console, on a regular file (accessed through NFS) or on a terminal window of the host system. The main thread of a dynamically loaded actor has its scheduling policy, priority and stack size set according to system-wide tunable parameters.

Actor Context

The precise context of an actor depends on how the system is configured. An extended actor has a file context similar to the file context of a UNIX process: it has a root directory as well as a current directory. It may also create, open, close, read and write files or sockets.

An extended actor runs on behalf of a user who is identified by means of a credentials structure. The actor credentials include: the identifier of the user, the identifier of the group of the user as well as a possibly empty list of identifiers of supplementary groups. Readers familiar with the concept of credentials in UNIX should note that the ChorusOS operating system concept of credentials is simpler than the UNIX one. ChorusOS 4.0 does not differentiate between real or effective user/group identification as it is not supported.

These actor credentials are used for file access. They are also used when the ChorusOS operating system runs in secured mode to check the validity of an operation. For example, in secured mode only the superuser, whose user identifier is 0, may load supervisor actors.

Standard Input/Output (I/O)

An extended actor may take advantage of the entire C library for dealing with I/O. In addition to the I/O interface provided by the C library, an extended actor may also use POSIX I/O services such as `open`, `read`, or `write`, as well as POSIX socket services such as `socket`, `bind`, and `connect`.

The following program may be run as an actor, and illustrates the way in which the C library might be used from an actor.

CODE EXAMPLE 5-2 Using the C Library from an Actor

```
#include <stdio.h>
#include <stdlib.h>
#include <chorus/stat.h>

#define BUF_SIZE 80

struct stat st;

int main(int argc, char** argv, char** envp)
{
    FILE* file;
    FILE* filew;
    char* buf;
    int res;

    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(1);
    }

    file = fopen(argv[1], "r");

    if (file == NULL) {
        fprintf(stderr, "Cannot open file %s\n", argv[1]);
        exit(1);
    }

    res = stat(argv[1], &st);
```



```

    if (res < 0) {
        fprintf(stderr, "Cannot stat file\n");
        exit(1);
    }

    printf("File size is %d mode 0x%x\n", st.st_size, st.st_mode);

    buf = (char*) malloc(BUF_SIZE);
    if (buf == NULL) {
        fprintf(stderr, "Cannot allocate buffer\n");
        exit(1);
    }

    bzero(buf, BUF_SIZE);
    res = read(fileno(file), buf, 80);
    if (res == -1) {
        fprintf(stderr, "Cannot read file\n");
        exit(1);
    }

    printf("%s\n", buf);

    if (argv[2] != NULL) {
        filew = fopen(argv[2], "w");

        if (filew == NULL) {
            fprintf(stderr, "Cannot open file %s\n", argv[2]);
            exit(1);
        }

        printf("Type any input you like: \n");

        do {
            scanf("%80s", buf);
            printf("buf=%s\n", buf);
            fprintf(filew, "%s", buf);
            printf("buf=%s\n", buf);
        } while (buf[0] != 'Q');
    }
    exit(0);
}

```

Note - This example assumes that `argv[0]` is valid, and the actor is linked with the `~lib/classix/libcx.a` library, since the library is common to both user and supervisor actors. Referencing `argv[0]` without checking if `argc` is greater than zero can cause the actor to make an exception and be deleted.

Allocating Memory

In any C program, memory can be dynamically allocated by means of the `malloc()` C library routine within actors, whether loaded at boot time or dynamically, and whether running as user or supervisor actors.

Code Example 5-2 shows a usage of the `malloc()` routine.

Terminating an Actor

As shown in the previous example, an actor may terminate by invoking the `exit()` routine, as with any typical C program. Invoking `exit()` ensures that all resources used by the actor are freed: I/O buffers will be flushed, all open files are closed, and all other system resources provided by features configured within the system are released automatically.

Spawning an Actor

So far, two ways of loading and running actors have been described: either the inclusion of the actor as part of the system image, or the usage of the `arun` mechanism. The ChorusOS operating system also enables an actor to dynamically spawn another actor from a binary file. This spawned actor may be either a supervisor or a user actor. This service is similar to the `exec()` UNIX system call:

```
#include <am/afexec.h>

int afexecve (const char* path,
              KnCap* actorCap,
              const AcParam* param,
              char* const* argv,
              char* const* envp);
```

This service creates a new actor whose capability is returned by the system at the location pointed to by the `actorCap` argument. The actor created will execute the binary file stored in the file named `path`. The main thread of this actor will run the main routine of the program. This thread will have the same scheduling attributes and stack size as an actor loaded using the `arun` mechanism.

`argv` and `envp` are pointers to the array of arguments and environments that will be received by the newly created actor.

The `afexec()` service comes in several variants, similar to the UNIX `exec()` call variants. When successful, all `afexec()` routines return the actor identifier of the

newly created actor. Otherwise, they return -1, and the error code is returned in the `errno` variable.

Most of the time, application writers will not need to use the `afexec()` service. They will use either the `arun` facility or include the actor as part of the system image. However, for convenience, some examples within this document do use this service.

Below is an example of use of the `afexecve()` service call.

- A user actor loaded by `arun` spawns another user actor, running the same executable file.
- They both print a trace and terminate.
- The first actor is distinguished from the one it spawns by the number of arguments: the first actor has no argument, but spawns the second actor passing it a string as its first argument. This string is printed by the spawned actor.
- The first actor prints the actor identifier of the spawned actor.

CODE EXAMPLE 5-3 Spawning an Actor

```
#include <stdio.h>
#include <errno.h>
#include <am/afexec.h>

AcParam param;
char*   spawnedArgs[3];
char*   tagPtr = "Welcome newly created actor!";

main(int argc, char** argv, char**envp)
{
    KnCap      spawnedCap;
    int        res;

    if (argc == 1) {
        /*
         * This is the first actor (or spawning actor):
         *   Binary file used to load this actor is passed
         *   by "arun" as argv[0],
         *
         *   Set an argument in order to enable the second
         *   actor to know it is the second one.
         */
        param.acFlags = AFX_USER_SPACE;

        spawnedArgs[0] = argv[0];
        spawnedArgs[1] = tagPtr;
        spawnedArgs[2] = NULL;
        /*
         * Other fields are implicitly set to NULL, as
         * param is allocated within the bss segment of
         * the program.
         */
        res = afexecve(argv[0], &spawnedCap, &param , spawnedArgs, envp);

        if (res == -1) {
```

```

        printf("Cannot spawn second actor, error %d\n", errno);
        exit(1);
    }
    printf("I succeeded creating actor whose aid is %d\n", res);
} else {
    /*
     * This is the spawned actor:
     * Check the number of args,
     * Print args,
     * Exit
     */
    if ((argc == 2) && (strcmp(tagPtr, argv[1]) == 0)) {
        /*
         * This is really the spawned actor.
         */
        printf("My spawning actor passed me this argument: %s\n",
            argv[1]);
    } else {
        printf("You ran %s with an argument, you should not!\n", argv[0]);
        exit(1);
    }
}
exit(0);
}

```

Note - This example assumes that `argv[0]` is valid, and the actor is linked with the `~lib/classix/libcx.a` library, since the library is common to both user and supervisor actors. Referencing `argv[0]` without checking if `argc` is greater than zero can cause the actor to make an exception and be deleted.

- A null `AcParam` argument instructs the system to use default values for `afexec()` calls.
- The `acFlags` field indicates, among other possibilities, whether the actor should be created as a user actor (when the flag is set to `AFX_USER_SPACE`), or as a supervisor actor (when the flag is set to `AFX_SUPERVISOR_SPACE`). These values are mutually exclusive: one and only one of the two values may be set. In addition, the user must make sure that the value of the flag is consistent with the binary file used to load the actor. Trying to create a supervisor actor with a binary file prepared for a user actor, by linking with the user libraries, will result in an error.

The `AFX_ANY_SPACE` option can be passed to instruct the operating system to retrieve the privilege of the binary file and create an actor with the same privilege.

- Unused fields of the `AcParam` argument must be set to 0.

Multithreaded Programming with the ChorusOS Operating System

This chapter describes how to use ChorusOS operating system services to create a multithreaded actor. It contains the following sections:

- “Basic Multi-Thread Programming” on page 133 is an overview of the multithreading model of the ChorusOS operating system.
- “Thread Handling” on page 135 explains how to identify, create, and delete a thread.
- “Synchronizing Threads” on page 142 explains the available methods for synchronizing threads.
- “Basic Scheduling Control” on page 150 explains how to schedule threads.
- “Managing Per-Thread Data” on page 154 explains how to maintain and use per-thread and shared data.
- “Threads and Libraries” on page 158 explains how to use libraries within a multithreaded actor.

Basic Multi-Thread Programming

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 6-1.

Threads may be created and deleted dynamically. A thread may be created in another actor 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 named 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 which are covered in a later section of this document.

Threads are scheduled by the kernel 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.

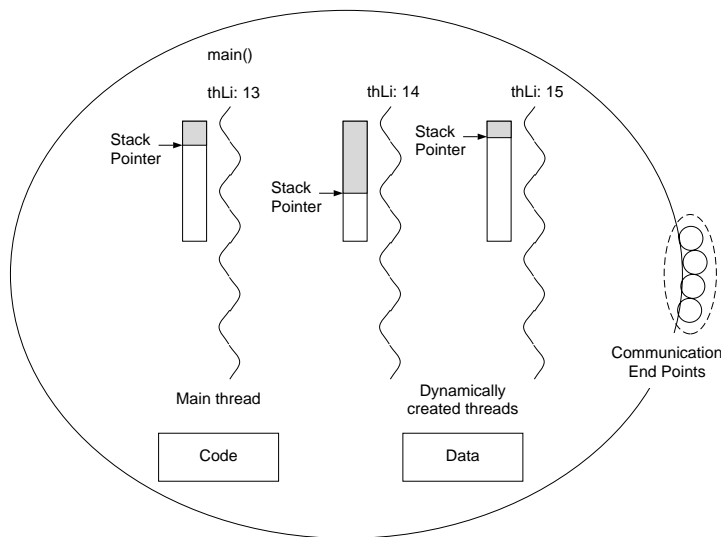


Figure 6-1 A Multi-Threaded Actor

Thread Handling

Getting a Thread Identifier

A thread may obtain its local identifier by means of the following ChorusOS operating system service:

```
#include <chorus.h>

int threadSelf();
```

An example of how this call can be used is provided in Code Example 6–1.

Creating a Thread

A thread may be created dynamically by means of the following ChorusOS operating system service:

```
#include <chorus.h>

int threadCreate(KnCap*      actorCap,
                 KnThreadLid* thLi,
                 KnThreadStatus status,
                 void*        schedParam,
                 void*        startInfo);
```

The `actorCap` parameter identifies the actor in which the new thread will be created. You can create the new thread in the current actor by passing `K_MYACTOR` as the actor capability. This is the usual case. Should this be successful, the local identifier of the newly created thread is returned at the location defined by the `thLi` parameter.

The `schedParam` parameter is used to define the scheduling properties of the thread to be created. If this parameter is set to 0, the created thread inherits the scheduling attributes of the creator thread.

The `startInfo` parameter is used to define the initial state of the thread, such as the initial program counter of the thread (the thread entry point), as well as the initial value of the stack pointer to be used by the created thread. You can also define whether the thread will run as a user thread or as a supervisor thread.

A thread needs a stack to run, in order to have room to store its local variables. When the thread is a user thread, the user must explicitly provide a stack to the thread. However, stacks for supervisor threads are implicitly allocated by the system. In fact, a system stack is allocated for all threads, even those running in user mode.

Note - As the operating system does not prevent the user stack from overflowing, checks must be made every time a thread is created.

System stacks are not allowed to overflow as memory will become corrupted, resulting in unpredictable operating system behavior.

Code Example 6-1 is a simple program illustrating the creation of a thread by the main thread of an actor. The actor is loaded by the `arun` command. Its main thread is implicitly created by the system. The goal of the example is to:

- create a thread, which prints a message, including its thread identifier
- simultaneously, the main thread prints another message with both thread identifiers
- the main thread then terminates the actor

This example will work without modification whether it is run as a user or as a supervisor actor. In the first case, a user thread must be created, while in the second case a supervisor thread must be created. Using the `actorPrivilege()` service call might be helpful for this purpose.

This example requires some kind of synchronization between the main thread and the created one. Execution of a thread can be suspended for a given delay:

```
#include <chorus.h>

int threadDelay(KnTimeVal* waitLimit);
```

This call suspends the execution of the invoking thread for a period specified by the `KnTimeVal` structure (see “Current Time ” on page 192 for more detail). There are two predefined values:

- `K_NOTIMEOUT` specifies an infinite delay.
- `K_NOBLOCK`, which specifies no delay. This is an explicit request for the processor to yield and reschedule another thread of the same priority.

These values may be used instead of the pointer to the `KnTimeVal` data structure. There is also a predefined macro which sets such a structure from a delay expressed in milliseconds: `K_MILLI_TO_TIMEVAL(KnTimeVal* waitLimit, int delay)`. For more information, see the `threadCreate(2K)`, `threadDelay(2K)`, and `threadSelf(2K)` man pages.

CODE EXAMPLE 6-1 Creating a Thread

```
(file: progov/thCreate.c)

#include <stdio.h>
#include <stdlib.h>
```



```

#include <chorus.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

int
childCreate(KnPc entry)
{
    KnActorPrivilege    actorP;
    KnDefaultStartInfo_f startInfo;
    char*               userStack;
    int                 childLid = -1;
    int                 res;

    /* Set defaults startInfo fields */
    startInfo.dsType      = K_DEFAULT_START_INFO;
    startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE;

    /* Get actor's privilege */
    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }

    /* Set thread privilege */
    if (actorP == K_SUPACTOR) {
        startInfo.dsPrivilege = K_SUPTHREAD;
    } else {
        startInfo.dsPrivilege = K_USERTHREAD;
    }

    /* Allocate a stack for user threads */
    if (actorP != K_SUPACTOR) {
        userStack = malloc(USER_STACK_SIZE);
        if (userStack == NULL) {
            printf("Cannot allocate user stack\n");
            exit(1);
        }
    }

    startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
}

/* Set entry point for the new thread */
startInfo.dsEntry = entry;

/* Create the thread in the active state */
res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
if (res != K_OK) {
    printf("Cannot create the thread, error %d\n", res);
    exit(1);
}

return childLid;
}

void
sampleThread()
{
    int myThreadLi;

```

```

myThreadLi = threadSelf();

printf("I am the new thread. My thread identifier is: %d\n", myThreadLi);

/* Block itself for ever */
threadDelay(K_NOTIMEOUT);
}

int main(int argc, char** argv, char**envp)
{
    int          myThreadLi;
    int          newThreadLi;
    int          res;
    KnTimeVal    wait;

    newThreadLi = childCreate((KnPc)sampleThread);

    myThreadLi = threadSelf();

    /* Initialize KnTimeVal structure */
    K_MILLI_TO_TIMEVAL(&wait, 10);

    /*
     * Suspend myself for 10 milliseconds to give the newly
     * created thread the opportunity to run before
     * the actor terminates.
     */
    res = threadDelay(&wait);

    printf("Parent thread identifier = %d, Child thread identifier = %d\n",
        myThreadLi, newThreadLi);

    return 0;
}

```

- The `schedParam` parameter is set to 0. As a result, the created thread will inherit the scheduling attributes of the creator thread.
- Note the usage of the `actorPrivilege()` service which enables the program to determine whether it must allocate a user stack area for the created thread or not, as well as to indicate the type of thread to be created.
- If the actor is a supervisor actor, the following line:
`startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE` only gives an indication to the system of the expected usage of the system stack. The maximum system stack length is defined by a global tunable value.
- On some platforms the stack pointer value passed in `dsUserStackPointer` is automatically decremented by the kernel before being used for the thread. This is done either to enforce the platform-required alignment, on 8 or 16 byte boundaries for example, or to reserve a space which will be accessed by a typical C language

routine because of the platform-specific calling conventions, such as saving the return address to the caller.

- The `status` parameter is used to create the thread in the active state, so that the thread is ready to execute as soon as it is created.
- Be aware that, although this program explicitly creates only one thread, there are in fact two threads running in this actor: the main thread created implicitly by the system when the actor is loaded, and the thread explicitly created by the program.
- The above example uses a service named `threadDelay()`, which allows a thread to suspend its execution for a certain period. The parent thread suspends itself for ten milliseconds, so that the child thread is able to run before `exit` is called. Without this suspension period in the parent thread, the actor could terminate before the created thread has run.

As explained earlier, the termination of an actor implies that all its resources are freed. Threads are not an exception to that rule. Thus, the `exit()` call at the end of the main routine will lead to the destruction of both threads. Use of the period within the parent thread is not a guarantee. Depending on the load of the system, ten milliseconds might not be sufficient to ensure that the child thread has completed its task. The `threadDelay()` has only been used in this example for the sake of simplicity, and is not recommended in practice for synchronizing threads. A more reliable synchronization scheme should be used to be sure that the actor does not terminate before the second thread has completed all jobs. These synchronization mechanisms are explained in “Synchronizing Threads” on page 142.

- The child thread uses the `K_NOTIMEOUT` special value to suspend itself for ever. This is a simple way to avoid undesirable behavior of the child thread until the actor terminates. Assume this call to `threadDelay()` does not exist. The child thread, after having executed the `printf()` statement, would reach the end of the `sampleThread()` routine, which being written in C terminates with a return instruction. However, the child thread has nowhere to return. As a result it would return to an unspecified location, probably resulting in a memory fault.

The system does not preset the stack of a thread to ensure that the thread is deleted upon return from its starting routine. You, the ChorusOS operating system programmer, must ensure that threads are properly cleaned up after they finish running. Mechanisms for coping with these types of situations are described in Chapter 7.

Deleting a Thread

A thread may be dynamically deleted by itself or by another one using the following service:

```
#include <chorus.h>

int threadDelete(KnCap* actorCap,
                KnThreadLid thLi);
```

This call enables a thread to delete another one inside the same actor, when `actorCap` is set to `K_MYACTOR`, by knowing the thread identifier of the thread to be deleted. It also enables a thread to delete another one inside another actor (provided they are both running on the same machine), as long as it provides both the actor capability and the target thread identifier. The predefined thread identifier `K_MYSELF` enables a thread to name itself without knowing its actual thread identifier.

Code Example 6-2 is a slightly different version of the previous program. The subroutine `childCreate()` is unchanged, but now the created thread kills itself, instead of going idle forever.

Note - This does not solve the synchronization problem occurring in the previous example: the main thread still does not know exactly when to terminate the actor.

Refer to the `threadDelete(2K)` man page.

CODE EXAMPLE 6-2 Deleting a Thread

```
(file: progov/thDelete.c)

#include <stdio.h>
#include <stdlib.h>

#include <chorus.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

int
childCreate(KnPc entry)
{
    KnActorPrivilege    actorP;
    KnDefaultStartInfo_f startInfo;
    char*               userStack;
    int                 childLid = -1;
    int                 res;

    startInfo.dsType      = K_DEFAULT_START_INFO;
    startInfo.dsStackSize = K_DEFAULT_STACK_SIZE;

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }

    if (actorP == K_SUPACTOR) {
        startInfo.dsPrivilege = K_SUPTHREAD;
    } else {
```

```

    startInfo.dsPrivilege = K_USERTHREAD;
}

if (actorP != K_SUPACTOR) {
    userStack = malloc(USER_STACK_SIZE);
    if (userStack == NULL) {
        printf("Cannot allocate user stack\n");
        exit(1);
    }
    startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
}

startInfo.dsEntry = entry;

res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
if (res != K_OK) {
    printf("Cannot create the thread, error %d\n", res);
    exit(1);
}

return childLid;
}

void
sampleThread()
{
    int myThreadLi;

    myThreadLi = threadSelf();

    printf("I am the new thread. My thread identifier is: %d\n", myThreadLi);

    /* Suicide */
    threadDelete(K_MYACTOR, K_MYSELF);

    /* Should never reach this point! */
}

int main(int argc, char** argv, char**envp)
{
    int myThreadLi;
    int newThreadLi;
    int res;
    KnTimeVal wait;

    newThreadLi = childCreate((KnPc)sampleThread);

    myThreadLi = threadSelf();

    /* Initialize KnTimeVal structure */
    K_MILLI_TO_TIMEVAL(&wait, 10);

    /*
     * Suspend myself for 10 milliseconds to give the newly
     * created thread the opportunity to run before
     * the actor terminates.
     */
    res = threadDelay(&wait);

    printf("Parent thread identifier = %d, Child thread identifier = %d\n",

```

```

myThreadLi, newThreadLi);

return 0;
}

```

- The `exit()` function is used instead of the `threadDelete()` function in the main thread. Using `threadDelete()` would leave the actor in a passive situation, with no thread running within it. This implies that resources used by an actor are not freed when the last thread is deleted.
- In the case of a user thread, deleting a thread does not imply that the stack of the thread will be freed. If the user stack was allocated through a call to `malloc()`, it must be freed through a call to `free()`. This cannot be done by the thread itself, it must be done by another thread. In the above example, the actor is going to terminate, so there is no real need to do this because all resources used by the actor will be returned to the system. In the case of a supervisor thread, the ChorusOS operating system frees the system stack it had allocated at `threadCreate()` time.

Synchronizing Threads

The previous section explained the need for threads to be synchronized accurately, avoiding using delays which are difficult to tune and which depend on the load of the system. The ChorusOS operating system offers various tools for synchronizing threads:

- *Semaphores*, which are common counting semaphores that support the *P* and *V* operations. See “Semaphores” on page 143.
- *Mutexes*, which provide a convenient and efficient way to implement mutual exclusion between multiple threads, in order to prevent a critical section from being executed in parallel by different threads. See “Mutexes” on page 147.
- *Thread semaphores*, which may be used to block a single thread awaiting the arrival of an event.
- *Event flags*, which may be useful when a thread has to handle multiple events, providing the kind of multiplexing which is offered by the `select` system call, but at a much lower level.

Semaphores

A semaphore is an integer counter associated with a queue, possibly empty, of waiting threads. At initialization, the semaphore counter receives a user-defined positive or null value. Initialization is performed by invoking the following ChorusOS operating system service:

```
#include <chorus.h>

int semInit(KnSem*      semaphore,
            unsigned int count);
```

The `semaphore` parameter is the location of the semaphore and `count` is the semaphore counter. The semaphore must have been previously allocated by the user: allocation is not performed by the ChorusOS operating system. This implies that semaphores may be freely allocated by the user where convenient for his applications. As data structures representing semaphores are allocated by the applications, the ChorusOS operating system does not impose any limit on the maximum number of semaphores which may be used within the system.

Two atomic operations, named P and V, are provided on these semaphores.

```
#include <chorus.h>

int semP(KnSem*      semaphore,
         KnTimeVal* waitLimit);
```

`semP()` decrements the counter by one. If the counter reaches a negative value, the invoking thread is blocked and queued within the semaphore queue. Otherwise the thread continues its execution normally. The `waitLimit` parameter may be used to control how long the thread will stay queued. If `waitLimit` is set to `K_NOTIMEOUT`, the thread will stay blocked until the necessary V operation is performed. In the case of the thread being awakened due to the expiration of the period, a specific error code is returned as the result of the `semP()` invocation. In this case, the counter is incremented to compensate for the effect of the `semP()` operation.

```
#include <chorus.h>

int semV(KnSem* semaphore);
```

`semV()` increments the counter by one. If the counter is still lower than or equal to zero, one of the waiting threads is picked up from the queue and awakened. If the counter is strictly greater than zero, there should be no thread waiting in the queue.

Figure 6–2 shows an example of two threads synchronizing by means of a semaphore.

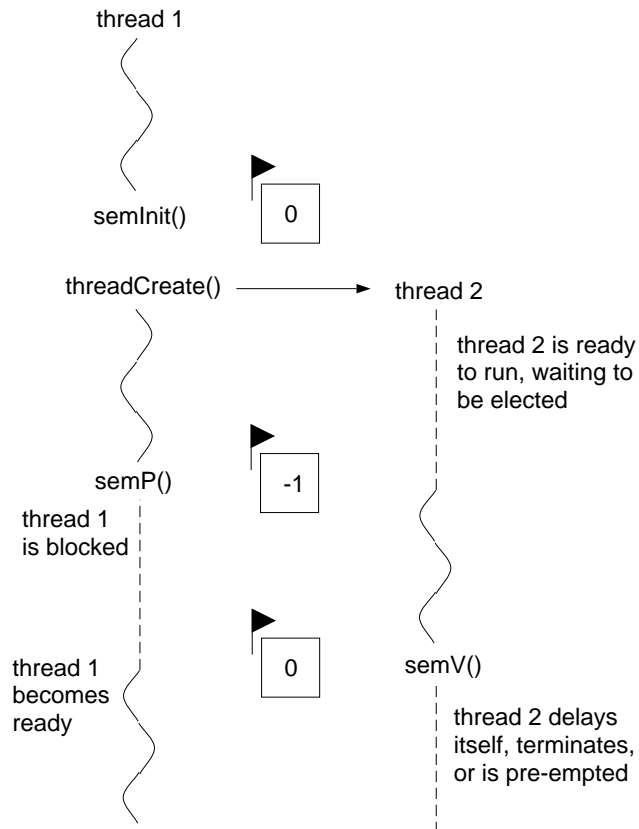


Figure 6-2 Two Threads Synchronizing with a Semaphore

The following example is based on the previous one, but the two threads explicitly synchronize by means of a semaphore, so that the actor will eventually be destroyed when the created thread has done its job and as soon as it has done so. Refer to the `semInit(2K)` man page.

CODE EXAMPLE 6-3 Synchronizing Using Semaphores

```

(file: progov/semaphore.c)

#include <stdio.h>
#include <stdlib.h>
#include <chorus.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

KnSem  sampleSem; /* Semaphore allocated as global variable */

int
childCreate(KnPc entry)
{
    KnActorPrivilege  actorP;
    KnDefaultStartInfo_f  startInfo;

```



```

char*                userStack;
int                  childLid = -1;
int                  res;

startInfo.dsType      = K_DEFAULT_START_INFO;
startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE;

res = actorPrivilege(K_MYACTOR, &actorP, NULL);
if (res != K_OK) {
    printf("Cannot get the privilege of the actor, error %d\n", res);
    exit(1);
}

if (actorP == K_SUPACTOR) {
    startInfo.dsPrivilege = K_SUPTHREAD;
} else {
    startInfo.dsPrivilege = K_USERTHREAD;
}

if (actorP != K_SUPACTOR) {
    userStack = malloc(USER_STACK_SIZE);
    if (userStack == NULL) {
        printf("Cannot allocate user stack\n");
        exit(1);
    }
    startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
}

startInfo.dsEntry = entry;

res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
if (res != K_OK) {
    printf("Cannot create the thread, error %d\n", res);
    exit(1);
}

return childLid;
}

void
sampleThread()
{
    int myThreadLi;
    int res;

    myThreadLi = threadSelf();

    printf("I am the new thread. My thread identifier is: %d\n", myThreadLi);

    res = semV(&sampleSem);
    if (res != K_OK){
        printf("Cannot perform the semV operation, error %d\n", res);
        exit(1);
    }

    /* Suicide */
    res = threadDelete(K_MYACTOR, K_MYSELF);
    if (res != K_OK){
        printf("Cannot suicide, error %d\n", res);
        exit(1);
    }
}

```

```

        /* Should never reach this point! */
    }

int main(int argc, char** argv, char**envp)
{
    int      myThreadLi;
    int      newThreadLi;
    int      res;

    /*
     * Initialize the semaphore to 0 so that
     * the first semP() operation blocks.
     */
    res = semInit(&sampleSem, 0);
    if (res != K_OK) {
        printf("Cannot initialize the semaphore, error %d\n", res);
        exit(1);
    }

    newThreadLi = childCreate((KnPc)sampleThread);

    myThreadLi = threadSelf();

    printf("Parent thread identifier = %d, Child thread identifier = %d\n",
        myThreadLi, newThreadLi);

    /*
     * Since semaphore has been initialized to 0
     * this semP will block until a semV is performed
     * by the created thread, letting the main thread know
     * that created thread's job is done.
     */
    res = semP(&sampleSem, K_NOTIMEOUT);
    if (res != K_OK) {
        printf("Cannot perform the semP operation, error %d\n", res);
        exit(1);
    }

    /*
     * Created thread has run and done all of its job.
     * It is time to safely exit.
     */
    return 0;
}

```

- The semaphore `sampleSem` is allocated as global data of the actor. As the address space of the actor is shared by all threads running within the actor, both threads can freely access the semaphore in order to synchronize.
- Avoid performing the semaphore initialization after having created the child thread. Depending on the scheduling, the second thread may start its execution as soon as it is created, and could reach the `semV()` operation before the semaphore has been initialized. Although the `semV()` could appear to work, `semP()` will never return due the fact that `semInit()` would reset the counter to 0.

- The synchronization will work whatever the order in which the `semP()` and `semV()` operations are done. If `semP()` is done first, the counter will be set to -1 and the main thread will be blocked. The `semV()` will awake the main thread. If scheduling is reversed, the `semV()` will set the counter to 1, so that when the `semP()` operation occurs, the counter will be decremented to 0, but the thread will not block.

Mutexes

Assume that the two threads need to access one or more global variables in a consistent fashion. A simple example could be that each of the threads needs to add two numbers to a unique global counter. Whatever the scheduling may be, the unique global counter should always reflect the accurate sum of all numbers added by both threads.

This could be done using semaphores. However, the ChorusOS operating system provides mutexes which have been specifically designed and tuned for these types of needs.

A mutex is a binary flag associated with a queue, possibly empty, of waiting threads. The mutex can be locked or free. At initialization, the mutex is set to the free state.

```
#include <chorus.h>

int mutexInit(KnMutex* mutex);
```

As for semaphores, the mutex must have been previously allocated by the user. This implies that mutexes may be allocated where convenient for the application, and that there is no limit imposed by the system on the maximum number of mutexes.

Three operations are provided on these mutexes.

- `mutexGet()` acquires the mutex: if the mutex is free, it is atomically locked and the thread continues its execution.

```
#include <chorus.h>

int mutexGet(KnMutex* mutex);
```

If the mutex is locked when the `mutexGet()` operation is invoked, the thread is blocked and queued in the list of threads, waiting for the mutex to become free. Note that there is no way to limit the time during which a thread waits to acquire a mutex.

- `mutexRel()` releases the mutex, returning it to its free state. If threads are blocked while waiting for the mutex, one of them is picked up from the list and activated with the mutex locked.

```
#include <chorus.h>

int mutexRel(KnMutex* mutex);
```

- The last operation is similar to `mutexGet()`, but does not block if the mutex is already locked.

```
#include <chorus.h>

int mutexTry(KnMutex* mutex);
```

By checking the return value of `mutexTry()`, you can determine whether the mutex was free and has been acquired by the current thread, or whether the mutex was already locked, in which case the operation has failed.

The following example shows a small and simple library routine named `sampleAdd()` which receives two integer arguments and adds them to a global variable one after the other. The code of the previous semaphore example has been modified so that both the main thread and the created thread perform a number of calls to that library. When the job is done, the main thread prints the result and terminates the actor. Refer to the `mutexInit(2K)` man page.

CODE EXAMPLE 6-4 Protecting Shared Data Using Mutexes

```
(file: progov/mutex.c)

#include <stdio.h>
#include <stdlib.h>
#include <chorus.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

KnSem sampleSem;
KnMutex sampleMutex;
long grandTotal;

int
childCreate(KnPc entry)
{
    KnActorPrivilege actorP;
    KnDefaultStartInfo_f startInfo;
    char* userStack;
    int childLid = -1;
    int res;

    startInfo.dsType = K_DEFAULT_START_INFO;
    startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE;

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }
}
```

```

    if (actorP == K_SUPACTOR) {
        startInfo.dsPrivilege = K_SUPTHREAD;
    } else {
        startInfo.dsPrivilege = K_USERTHREAD;
    }

    if (actorP != K_SUPACTOR) {
        userStack = malloc(USER_STACK_SIZE);
        if (userStack == NULL) {
            printf("Cannot allocate user stack\n");
            exit(1);
        }
        startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
    }

    startInfo.dsEntry = entry;

    res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
    if (res != K_OK) {
        printf("Cannot create the thread, error %d\n", res);
        exit(1);
    }

    return childLid;
}

void
sampleAdd(int a, int b)
{
    int    res;

    res = mutexGet(&sampleMutex);

    grandTotal += a;
    grandTotal += b;

    res = mutexRel(&sampleMutex);
}

void
sampleThread()
{
    int res;
    int i;

    for(i = 0; i < 10; i++) {
        sampleAdd(threadSelf(), i);      /* Why not ??? */
    }

    res = semV(&sampleSem);
    if (res != K_OK){
        printf("Cannot perform the semV operation, error %d\n", res);
        exit(1);
    }

    /* Suicide */
    threadDelete(K_MYACTOR, K_MYSELF);
}

int main(int argc, char** argv, char**envp)
{

```

```

int      i;
int      newThreadLi;
int      res;

res = semInit(&sampleSem, 0);
if (res != K_OK) {
    printf("Cannot initialize the semaphore, error %d\n", res);
    exit(1);
}

res = mutexInit(&sampleMutex);

newThreadLi = childCreate((KnPc)sampleThread);

for(i = 0; i < 20; i++){
    sampleAdd(threadSelf(), i);      /* Why not ??? */
}

res = semP(&sampleSem, K_NOTIMEOUT);
if (res != K_OK) {
    printf("Cannot perform the semP operation, error %d\n", res);
    exit(1);
}

printf("grandTotal is %d\n", grandTotal);

return 0;
}

```

- The mutex is allocated within the global data of the actor and is initialized before it is ever used.
- The `sampleAdd()` routine uses the mutex to protect access to the `grandTotal` variable and make it atomic. Note that the `mutexGet()` and `mutexRel()` operations perform the bulk of the work. Mutex operations should always be used in pairs, as in this example.
- A mutex is not recursive, a thread which has locked a mutex will deadlock if it tries to perform a second `mutexGet()` operation on the same mutex.

Basic Scheduling Control

The ChorusOS operating system provides two alternative ways of scheduling threads. These two features are mutually exclusive:

- either the ChorusOS operating system is configured with the default scheduler,
- or it is configured with the `ROUND_ROBIN` feature.

The default FIFO scheduler defines a pure priority-based, preemptive, FIFO (first-in first-out) policy. Priority of threads may vary from `K_FIFO_PRIOMAX` (0 and highest priority) to `K_FIFO_PRIOMIN` (255 and lowest priority). Within this policy, a thread which 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 becomes ready to run. A preempted thread is placed at the head of its priority queue, so that it will be selected when no other ready thread has a greater priority.

The `ROUND_ROBIN` feature is a general framework supporting simultaneous multiple scheduling policies or classes. The main classes dealt with here are the `CLASS_FIFO` and the `CLASS_RR` policies.

The `CLASS_FIFO` reproduces the behavior of the default scheduler policy precisely.

The `CLASS_RR` implements a priority-based preemptive policy with round-robin time slicing. Priority of threads may vary from `K_RR_PRIOMAX` to `K_RR_PRIOMIN`. It is similar to the default scheduler policy, except that an elected thread is given a fixed time quantum. If the thread is still running at quantum expiration, it is de-scheduled and placed at the end of its priority queue, thus yielding the processor to other threads of equal priority (if any).

It is possible to set scheduling attributes of threads at thread creation time (using the `void* schedParams` parameter of `threadCreate()`). It is also possible to get and modify scheduling attributes of a thread dynamically through the following call.

```
#include <chorus.h>
#include <sched/chFifo.h>
#include <sched/chRr.h>
#include <sched/chRt.h>
#include <sched/chTs.h>

int threadScheduler(KnCap*      actorCap,
                   KnThreadLid thLi,
                   void*        oldParam,
                   void*        newParam);
```

This service enables you to get or set scheduling parameters of any thread of any actor, as long as both the actor capability and the thread identifier are known. `threadScheduler()` returns the current scheduling attributes of the target thread at the location defined by `oldParam`, if non-null. It will also set the attributes of the target thread according to the description provided at the location defined by `newParam` if non-null.

As the size, layout and semantics of scheduling parameters may vary depending on the scheduler configured in the system, or on the class of the `ROUND_ROBIN` framework, parameters are untyped in the generic interface definition. However, all scheduling parameter descriptions are similar, at least for the initial fields:

```

struct KnFifoThParms {
    KnSchedClass    fifoClass;        /* Always set to K_SCHED_FIFO */
    KnFifoPriority    fifoPriority;
} KnFifoThParms;

struct KnRrThParms {
    KnSchedClass    rrClass;          /* Always set to K_SCHED_RR */
    KnRrPriority     rrPriority;
} KnRrThParms;

```

The first field defines the scheduling policy applied or to be applied to the thread. The second field defines the priority of the thread within the scheduling policy.

Code Example 6-5 is based on the semaphore example, with a modification to the `childCreate()` routine so that it can receive scheduling attributes of the thread to be created. The main thread invokes this modified routine, so that the created thread will start as soon as it is created, rather than waiting for the main thread to yield the processor. Thus, the created thread must be given a higher priority than the main thread.

Refer to the `threadScheduler(2K)` and `threadCreate(2K)` man pages.

CODE EXAMPLE 6-5 Changing Scheduling Attributes

```

(file: progov/thSched.c)

#include <stdio.h>
#include <stdlib.h>
#include <chorus.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

KnSem    sampleSem;

int
childSchedCreate(KnPc entry, void* schedParams)
{
    KnActorPrivilege    actorP;
    KnDefaultStartInfo_f startInfo;
    char*               userStack;
    int                 childLid = -1;
    int                 res;

    /* Set defaults startInfo fields */
    startInfo.dsType      = K_DEFAULT_START_INFO;
    startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE;

    /* Get actor's privilege */
    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }

    /* Set thread privilege */

```



```

if (actorP == K_SUPACTOR) {
    startInfo.dsPrivilege = K_SUPTHREAD;
} else {
    startInfo.dsPrivilege = K_USERTHREAD;
}

    /* Allocate a stack for user threads */
if (actorP != K_SUPACTOR) {
    userStack = malloc(USER_STACK_SIZE);
    if (userStack == NULL) {
        printf("Cannot allocate user stack\n");
        exit(1);
    }

    startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
}

    /* Set entry point for the new thread */
startInfo.dsEntry = entry;

    /* Create the thread in the active state */
res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, schedParams, &startInfo);
if (res != K_OK) {
    printf("Cannot create the thread, error %d\n", res);
    exit(1);
}

return childLid;
}

void
sampleThread()
{
    int myThreadLi;
    int res;

    myThreadLi = threadSelf();

    printf("I am the new thread. My thread identifier is: %d\n", myThreadLi);

    res = semV(&sampleSem);
    if (res != K_OK){
        printf("Cannot perform the semV operation, error %d\n", res);
        exit(1);
    }
    threadDelete(K_MYACTOR, K_MYSELF);
}

int main(int argc, char** argv, char**envp)
{
    int myThreadLi;
    int newThreadLi;
    int res;
    KnThreadDefaultSched schedParams;

    res = semInit(&sampleSem, 0);
    if (res != K_OK) {
        printf("Cannot initialize the semaphore, error %d\n", res);
        exit(1);
    }

```

```

}

/* acquire my own scheduling attributes */
res = threadScheduler(K_MYACTOR, K_MYSELF, &schedParams, NULL);

/* Increase priority of thread to be created */
schedParams.tdPriority -= 1;

newThreadLi = childSchedCreate((KnPc)sampleThread, &schedParams);

myThreadLi = threadSelf();

printf("Parent thread identifier = %d, Child thread identifier = %d\n",
myThreadLi, newThreadLi);

res = semP(&sampleSem, K_NOTIMEOUT);
if (res != K_OK) {
    printf("Cannot perform the semP operation, error %d\n", res);
    exit(1);
}

return 0;
}

```

- First, the main thread needs to get its own scheduling attributes. As these are not known, a `KnThreadDefaultSched` structure is used as the output argument of the call to `threadScheduler()`. The last argument of `threadScheduler()` is set to null as the current scheduling attributes of the main thread wish to be preserved.
- In order to give a higher priority to the created thread, decrease the numerical value of the priority. Increasing the priority value has the reverse effect.

Managing Per-Thread Data

One of the most common issues in a multithreaded environment is how to manage per-thread data structures. This may become an important question for libraries. In a single-threaded process, managing these data as global variables is fine. In a multithreaded environment, it will no longer work.

The ChorusOS operating system provides a convenient way for threads to manage per-thread data. A piece of data which needs to be instantiated on a per-thread basis must be associated with a unique key. The key may be obtained from the system through a call to `ptdKeyCreate()`. This data may be accessed using specific calls named `ptdSet()` and `ptdGet()`.

```
#include <pd/chPd.h>

int ptdKeyCreate(PdKey* key,
                KnPdHdl destructor);
```

`ptdKeyCreate()` generates a unique key, which is opaque to the user. This key is stored at the location defined by the `key` argument. The user may, optionally, specify a routine as the destructor argument. This routine will be invoked at thread deletion time and will be passed the value associated with `key`. Upon return from `ptdKeyCreate()`, the value associated with `key` is 0. This type of key is visible to all threads of the actor, but each thread using a given key will have its own private copy of the data.

```
#include <pd/chPd.h>

int ptdSet(PdKey key,
          void* value);
```

`ptdSet()` enables a thread to associate the value `value` with the key `key` which has been generated previously by a call to `ptdKeyCreate()`.

```
#include <pd/chPd.h>

int ptdGet(PdKey key);
```

`ptdGet()` returns the last value associated with the key by this same thread.

Code Example 6–6 includes a small library that returns a pointer to the next word of a string. This is a simplified version of the `strtok()` C library routine. For simplicity, it is assumed that words are always separated by spaces in the string.

This library is callable simultaneously from different threads, each thread working on its own string. The routine that returns the pointer to the next word does not take any parameters.

These routines are called `snw` routines (where `snw` stands for String Next Word). There is a `snwSet(char *str)` routine which defines the string that will be looked up by the invoking thread, and a `char* snwGet()` returning a pointer to the next word.

The library is invoked from the main thread and the created thread on two different strings in order to count the number of words in each string. The results are printed and the threads are synchronized before terminating the actor.

Refer to the `ptdKeyCreate(2K)`, `ptdSet(2K)`, and `ptdGet(2K)` man pages.

CODE EXAMPLE 6-6 Managing Per-Thread Data

```
(file: progov/perThreadData.c)

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <chorus.h>
#include <pd/chPd.h>

#define USER_STACK_SIZE (1024 * sizeof(long))

KnSem    sampleSem;
PdKey    snwKey;

int
childCreate(KnPc entry)
{
    KnActorPrivilege    actorP;
    KnDefaultStartInfo_f startInfo;
    char*               userStack;
    int                 childLid = -1;
    int                 res;

    startInfo.dsType      = K_DEFAULT_START_INFO;
    startInfo.dsSystemStackSize = K_DEFAULT_STACK_SIZE;

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get the privilege of the actor, error %d\n", res);
        exit(1);
    }

    if (actorP == K_SUPACTOR) {
        startInfo.dsPrivilege = K_SUPTHREAD;
    } else {
        startInfo.dsPrivilege = K_USERTHREAD;
    }

    if (actorP != K_SUPACTOR) {
        userStack = malloc(USER_STACK_SIZE);
        if (userStack == NULL) {
            printf("Cannot allocate user stack\n");
            exit(1);
        }
        startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
    }

    startInfo.dsEntry = entry;

    res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
    if (res != K_OK) {
        printf("Cannot create the thread, error %d\n", res);
        exit(1);
    }

    return childLid;
}

void
```

```

snwInit()
{
    int res;
    /* Just allocate a key for our "snw" library */
    res = ptdKeyCreate(&snwKey, NULL);
    if (res != K_OK) {
        printf("Cannot create a ptd key, error %d\n", res);
        exit(1);
    }
}

void
snwSet(char* str)
{
    int res;

    res = ptdSet(snwKey, str);
    if (res != K_OK) {
        printf("Cannot set the ptd key, error %d\n", res);
        exit(1);
    }
}

char*
snwGet()
{
    int res;
    char* p;
    char* s;

    p = (char*)ptdGet(snwKey);
    if (p == NULL) return NULL;

    s = strchr(p, ' ');

    if (s != NULL) {
        s++;
    } else if (*p != '\0') {
        /* Last word might not have a following space */
        s = p + strlen(p);
    }

    res = ptdSet(snwKey, s);
    return s;
}

void
sampleThread()
{
    char* ptr;
    int words = 0;
    int res;

    snwSet("This is the child thread!");

    for (ptr= snwGet(); ptr != NULL; ptr = snwGet()) {
        words++;
    }

    printf("Child thread found %d words.\n", words);
}

```

```

    res = semV(&sampleSem);
    if (res != K_OK){
        printf("Cannot perform the semV operation, error %d\n", res);
        exit(1);
    }
    threadDelete(K_MYACTOR, K_MYSELF);
}

int main(int argc, char** argv, char**envp)
{
    char*      ptr;
    int        words = 0;
    int        res;
    int        newThreadLi;

    res = semInit(&sampleSem, 0);
    if (res != K_OK) {
        printf("Cannot initialize the semaphore, error %d\n", res);
        exit(1);
    }

    snwInit();

    newThreadLi = childCreate((KnPc)sampleThread);

    snwSet("I am the main thread and counting words in this string!");

    for (ptr= snwGet(); ptr != NULL; ptr = snwGet()) {
        words++;
    }

    printf("Main thread found %d words.\n", words);

    res = semP(&sampleSem, K_NOTIMEOUT);
    if (res != K_OK) {
        printf("Cannot perform the semP operation, error %d\n", res);
        exit(1);
    }

    return 0;
}

```

Threads and Libraries

As illustrated in the previous example, it is often the case that C and C++ libraries have been designed for UNIX processes which were initially mono-threaded entities. In order to allow C programmers to continue using the usual libraries within multithreaded actors, the ChorusOS operating environment provides a set of adapted

C libraries which may be used from different threads of a given actor without encountering problems.

In the previous examples, some of these adapted libraries, such as `printf()`, `fprintf()`, `fopen()`, and `malloc()`, were already used. All of these C libraries have been adapted to work efficiently even within a multithreaded actor. Modifications are not visible to the programmer. They rely mainly on synchronization such as mutexes for protecting critical sections and on the per-thread data mechanism to store per-thread global data.

Some libraries did not require any modification and can work in a straightforward fashion within a multithreaded actor. These libraries, such as `strtol()` (string to lower case), work exclusively on local variables and do not access or generate any global states.

Memory Management

Actors within the ChorusOS operating system environment may extend their address space using the `malloc()` library call as illustrated earlier in this document. However, this is a rather inflexible way of allocating memory, as there is no way to control the attributes of the allocated memory; that is, whether it is a read only or a read/write memory area. The `malloc()` routine uses the ChorusOS operating system services described in this section. These services may also be used to share a region of memory between two or more actors.

This chapter explains the recommended way of allocating memory for an actor. It contains the following sections:

- “Memory Region Descriptors” on page 161 explains how a memory region is identified and described.
- “Allocating and Freeing Memory Regions” on page 163 explains how to allocate and free memory.
- “Sharing Memory Between Two Actors” on page 167 explains how to share memory between actors.

The ChorusOS hot restart feature provides support for using persistent memory; memory which can extend beyond the lifetime of the runtime instance of an actor. Hot restart is not covered in this chapter. For information about using hot restart and the persistent memory services it provides, see the *ChorusOS 4.0 Hot Restart Programmer's Guide*.

Memory Region Descriptors

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. An area of memory to be allocated or freed is described to the system through a region descriptor of the following type:

```
typedef struct {
    VmAddr    startAddr;
    VmSize    size;
    VmFlags    options;
    VmAddr    endAddr;
    void*     opaque1;
    VmFlags    opaque2;
} KnRgnDesc;
```

The `startAddr` field defines the starting address of the memory region. The `size` field defines its length expressed in bytes. The `options` field enables a low level control on the attributes of the memory region to be allocated. The `opaque1` and `opaque2` fields should be set to `NULL` if they are not being used by the application.

The `options` field is a combination of flags, of which the following are the most important:

- `K_WRITABLE` tells the system that the memory region to be created must be writable, otherwise, the memory region will be read only.
- `K_FILLZERO` tells the system that the content of the memory region to be created must be filled with zeroes upon creation. If this flag is not set, the content of the memory region at creation time is unspecified.
- `K_ANYWHERE` tells the system that the actual address used to allocate the region is not critical to the application. An appropriate address will be selected by the system and returned to the application. This avoids the need for the application to find out which addresses are already in use within the actor address space. It also permits memory to be allocated within a library without any possible conflict with an existing address space.
- `K_SUPERVISOR` tells the system that the memory to be allocated will be part of the supervisor address space rather than of the user address space. This flag is usually set by actors running in supervisor mode.

On a ChorusOS operating system configured with the Virtual Memory feature, further options are available. The most important one allows control of the swapping policy to be applied to the pages of the created region:

- `K_NODEMAND` tells the system that no page fault should ever occur on such a memory region. Physical pages are allocated to the region at creation time and they will never be swapped out. Thus the region is locked in memory.

Allocating and Freeing Memory Regions

A memory region is allocated through the following call:

```
#include <chorus.h>

int rgnAllocate(KnCap* actorCap,
               KnRgnDesc* rgnDesc);
```

This call creates a memory region as described by the `rgnDesc` parameter within the address space of the actor defined by the `actorCap` parameter. Most applications set `actorCap` to `K_MYACTOR` to manage their own address space.

An unused part of an address space may be freed by the following call:

```
#include <chorus.h>

int rgnFree(KnCap* actorCap,
            KnRgnDesc* rgnDesc);
```

This call frees a memory region as described by the `rgnDesc` parameter within the address space of the actor defined by the `actorCap` parameter.

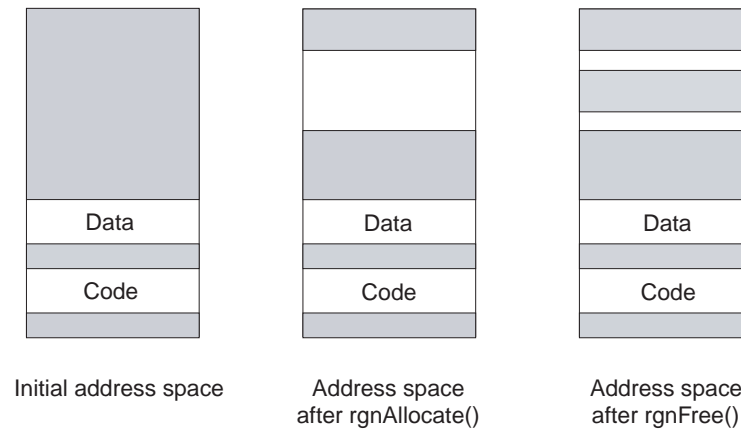


Figure 7-1 Memory Region Allocation and Deallocation

Code Example 7-1 does the following:

- Allocates a memory region with the `K_ANYWHERE` option.
- Retrieves the address of the allocated region and prints it.
- Copies a string to the beginning of the region.

- Creates a second region immediately preceding the first.
- Copies the string from the beginning of the first region to the beginning of the second region.
- Frees an area of memory spanning the junction between the two regions.
- Copies the string from the beginning of the second region to the lowest memory address still accessible outside the freed memory area.
- Ensures that the program is able to run in a user or supervisor actor.

The main steps of the example are illustrated in Figure 7-1. Refer to the `rgnAllocate(2K)` and `rgnFree(2K)` man pages.

CODE EXAMPLE 7-1 Allocating a Memory Region

```
(file: progov/rgnAlloc.c)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <chorus.h>

#define RGN_SIZE_1 (6 * vmPageSize())
#define RGN_SIZE_2 (3 * vmPageSize())
#define FREE_START (2 * vmPageSize())
#define FREE_SIZE (4 * vmPageSize())
#define STILL_ALLOC_START (FREE_SIZE - (RGN_SIZE_2 - FREE_START))

int main(int argc, char** argv, char**envp)
{
    KnRgnDesc      rgnDesc;
    KnActorPrivilege actorP;
    int            res;
    VmFlags        rgnOpt = 0;
    char*          ptr1 = NULL; /* Avoids "uninit'd" warning */
    char*          ptr2 = NULL; /* Avoids "uninit'd" warning */

    printf("Starting rgnAllocate example\n");

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get actor privilege, error %d\n", res);
        exit(1);
    }

    if (actorP == K_SUPACTOR) {
        rgnOpt = K_SUPERVISOR;
    }

    rgnDesc.size = RGN_SIZE_1;
    rgnDesc.options = rgnOpt | K_WRITABLE | K_FILLZERO | K_ANYWHERE;
    rgnDesc.opaque1 = NULL;
    rgnDesc.opaque2 = NULL;

    /*
     * No need to set rgnDesc.startAddr
    */
}
```

```

        * since we set the K_ANYWHERE flag
        */
res = rgnAllocate(K_MYACTOR, &rgnDesc);

if (res == K_OK) {
    printf("Successfully allocated memory starting at 0x%x\n",
        rgnDesc.startAddr);

    ptr1 = (char*) rgnDesc.startAddr;
} else {
    printf("First rgnAllocate failed with error %d\n", res);
    exit(1);
}

strcpy(ptr1, "Fill the allocated memory with this string\n");

/*
 * Second allocate has a fixed address, such that
 * both memory areas will be contiguous. Hence
 * we do not want the K_ANYWHERE flag any more.
 */
rgnDesc.size      = RGN_SIZE_2;
rgnDesc.options   &= ~K_ANYWHERE;
rgnDesc.startAddr -= RGN_SIZE_2;

res = rgnAllocate(K_MYACTOR, &rgnDesc);

if (res == K_OK) {
    printf("Successfully allocated memory starting at 0x%x\n",
        rgnDesc.startAddr);
    ptr2 = (char*) rgnDesc.startAddr;
} else {
    printf("Second rgnAllocate failed with error %d\n", res);
    exit(1);
}

/* Copy from first allocated area to second one */
strcpy(ptr2, ptr1);

/*
 * Free a memory area spanning both areas
 * previously created
 */
rgnDesc.options = NULL;
rgnDesc.startAddr = (VmAddr) (ptr2 + FREE_START);
rgnDesc.size      = FREE_SIZE;

res = rgnFree(K_MYACTOR, &rgnDesc);

if (res != K_OK) {
    printf("Cannot free memory, error %d\n", res);
    exit(1);
}

/*
 * Access to ptr2: beginning of secondly allocated area
 * is still valid.
 * Access to ptr1 is now invalid: memory has been freed.
 */
printf("%s", ptr2);

```

```

    /*
     * Access to "end" of first allocated area
     * is still valid
     */
    ptr1 += STILL_ALLOC_START;
    strcpy(ptr1, ptr2);

    /*
     * Remaining memory areas not yet freed will
     * effectively be freed at actor termination time.
     */
    return 0;
}

```

- Region descriptors are uniquely used to describe a creation or deletion operation on the system. They are not kept by the system in the way they are given to the `rgnAllocate()` call. As an example, allocation of two contiguous areas with the same attributes (writable, fill zero) and the same opaque fields will result in the system recognizing a single region, the size of which is the sum of the sizes passed as part of the two region descriptors.
- You cannot allocate a region on a range of addresses which are not free. No implicit deallocation of the address space is undertaken by the system, instead an error code `K_EOVERLAP` is returned to the caller.
- A call to `rgnFree()` does not need to reuse a region descriptor that was used to allocate a memory area. A free operation may freely span over several regions which were allocated by separate operations. Similarly, a free operation may only free a chunk of memory in the middle of a large memory area which was allocated in a single operation. See Figure 7-1.
- Only the precise region described in the region descriptor will be freed. The free operation is not extended to match the address range which was allocated at `rgnAllocate()` time.
- The options field of the region descriptor must be set to 0 for a free operation. Otherwise, you may set it to `K_FREEALL`, in which case all memory regions of the actor will be freed: the code, the data, the stacks. The `K_FREEALL` option should therefore be used with care.
- All memory areas which have been dynamically allocated are freed when the actor terminates.

Sharing Memory Between Two Actors

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. The memory area does not need to be located at the same address within the address space of each actor.

This mechanism is based on the following service:

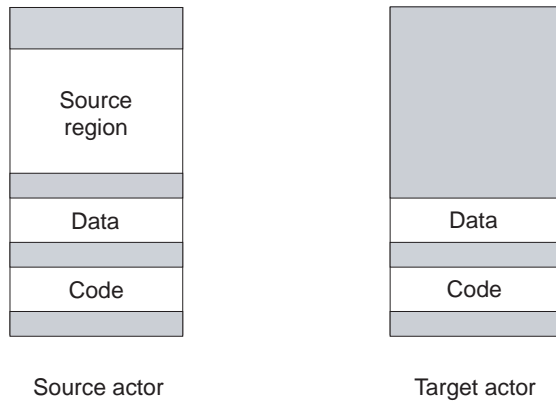
```
#include <chorus.h>

int rgnMapFromActor(KnCap*      targetActor,
                   KnRgnDesc*targetRgnDesc,
                   KnCap*      sourceActor,
                   KnRgnDesc*sourceRgnDesc);
```

This call allows mapping of a memory area as defined by `sourceActor` and `sourceRgnDesc` in the address space of the actor defined by the `targetActor` parameter. The source memory area is explicitly defined by the `startAddr` and `size` fields of the `sourceRgnDesc` parameter. The region created within the address space of the target actor is defined by the `targetRgnDesc` parameter: the address may be fixed or undefined if the `K_ANYWHERE` flag is set. After the mapping has been established, both actors may freely use the shared memory area.

Figure 7-2 shows two actors before they share memory and after sharing has been established. Usually some synchronization mechanism is required in order to get a consistent view of the memory: semaphores may be used in shared memory areas to synchronize threads from the various actors.

Source and target actors before sharing has been established



Source and target actors after sharing has been established

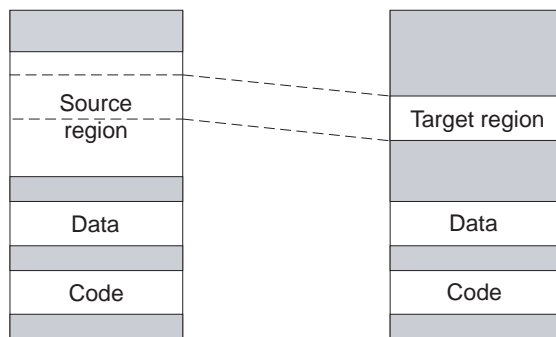


Figure 7-2 Actors Sharing Memory

Code Example 7-2 does the following:

- The actor started by an `arun` command allocates a memory region, then spawns an actor running the same executable file using the `afexecve()` call.
- The spawned actor uses the parameters set up by the spawning actor to establish a sharing of memory.
- Some data is passed through the shared memory from the spawning to the spawned actor. A semaphore allocated in the shared memory area, and initialized by the spawning actor, is used to synchronize both actors. Thus, the first actor will know when the spawned actor has changed the contents of the shared area.
- Both actors then terminate.
- The spawned actor should retrieve the information needed to establish the mapping from its arguments.

This example uses a call which enables an actor to get its own capability in order to pass it to another actor.

```
#include <chorus.h>

int actorSelf(KnCap* myCap);
```

Refer to the `rgnMapFromActor(2K)` man page.

CODE EXAMPLE 7-2 Sharing a Memory Region

```
(file: progov/rgnMapFromActor.c)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <chorus.h>
#include <am/afexec.h>

AcParam param;

#define RGN_SIZE (3 * vmPageSize())
#define SHARED_RGN_SIZE (1 * vmPageSize())

typedef struct sampleSharedArea {
    KnSem    sem;
    char     data[1];
} sharea_t;

char capString[80];
char sharedAddr[20];

int main(int argc, char** argv, char**envp)
{
    KnRgnDesc    rgnDesc;
    sharea_t*    ptr;
    KnCap        spawningCap;
    KnCap        spawnedCap;
    int          res;
    VmFlags      rgnOpt = 0;
    KnActorPrivilege actorP;

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get privilege, error %d\n", res);
        exit(1);
    }

    if (actorP == K_SUPACTOR) {
        rgnOpt = K_SUPERVISOR;
    }

    if (argc == 1) {
        /*
         * This is the first actor (or spawning actor):
         *   Allocate a memory region
         */
    }
}
```

```

        *   Initialize a semaphore within the region
        *   Spawn the second actor
        *   Wait on the semaphore
        *   Get data written in shared mem by spawned actor
        *   Terminate
    */
    rgnDesc.size = RGN_SIZE;
    rgnDesc.options = rgnOpt | K_ANYWHERE | K_WRITABLE | K_FILLZERO;
    rgnDesc.opaque1 = NULL;
    rgnDesc.opaque2 = NULL;

    res = rgnAllocate(K_MYACTOR, &rgnDesc);
    if (res != K_OK) {
        printf("Cannot allocate memory, error %d\n", res);
        exit(1);
    }

    ptr = (sharea_t*) rgnDesc.startAddr;

    strcpy(&ptr->data[0], "First actor initializing the shared mem\n");

    res = semInit(&ptr->sem, 0);

    /*
     * Get my own capability and pass it as a string argument
     * to spawned actor, so that it may use it to share memory
    */
    actorSelf(&spawningCap);
    sprintf(capString, "0x%x 0x%x 0x%x 0x%x", spawningCap.ui.uiHead,
        spawningCap.ui.uiTail, spawningCap.key.keyHead,
        spawningCap.key.keyTail);

    /*
     * Pass address of memory to be shared as a string argument
     * to spawned actor.
    */
    sprintf(sharedAddr, "0x%x", ptr);

    param.acFlags = (actorP == K_SUPACTOR)? AFX_SUPERVISOR_SPACE :
        AFX_USER_SPACE;
    res = afexeclp(argv[0], &spawnedCap, &param, argv[0], capString,
        sharedAddr, NULL);
    if (res == -1) {
        printf("cannot spawn second actor, error %d\n", errno);
        exit(1);
    }

    semP(&ptr->sem, K_NOTIMEOUT);

    printf("%s", &ptr->data[0]);

} else {

    KnRgnDesc srcRgn;
    KnRgnDesc tgtRgn;
    unsigned long uHead, uTail, kHead, kTail;
    /*
     * This is the spawned actor:
     * Get arguments
     * Set up the memory sharing

```

```

        *   Write some string in shared memory
        *   Wake up spawning actor
        *   Terminate
    */
    sscanf(argv[1], "0x%x 0x%x 0x%x 0x%x", &uHead, &uTail, &kHead, &kTail);
    spawningCap.ui.uiHead = uHead;
    spawningCap.ui.uiTail = uTail;
    spawningCap.key.keyHead = kHead;
    spawningCap.key.keyTail = kTail;

    sscanf(argv[2], "0x%x", &srcRgn.startAddr);

    if (actorP != K_SUPACTOR) {

        srcRgn.size      = SHARED_RGN_SIZE;
        tgtRgn.startAddr = srcRgn.startAddr;
        tgtRgn.size      = SHARED_RGN_SIZE;
        tgtRgn.options   = rgnOpt | K_WRITABLE;
        tgtRgn.opaque1   = NULL;
        tgtRgn.opaque2   = NULL;

        res = rgnMapFromActor(K_MYACTOR, &tgtRgn, &spawningCap, &srcRgn);

        if (res != K_OK) {
            printf("Cannot share memory, error %d\n", res);
            exit(1);
        }
        ptr = (sharea_t*) tgtRgn.startAddr;

    } else {
        /*
         * Both actors are running in supervisor space,
         * There is no need to perform the rgnMapFromActor.
         * One may use the received shared address.
         */
        ptr = (sharea_t*) srcRgn.startAddr;
    }

    /* Get data from spawning actor */
    printf("Spawning actor sent: %s", &ptr->data[0]);

    /* Modify contents of shared memory */
    sprintf(&ptr->data[0], "Spawned actor mapped shared memory at 0x%x\n",
        ptr);

    res = semV(&ptr->sem);
    if (res != K_OK) {
        printf("Spawned actor failed on semV, error %d\n", res);
        exit(1);
    }
}
return 0;
}

```

- Semaphores have been tuned to be highly optimized. There are thus some constraints on their use: they may be used freely in a region of shared memory

between two user actors, even though the shared area is not mapped at the same address in each actor. In supervisor space, a semaphore cannot be accessed using two different addresses. As supervisor address space is partitioned common space, there is no real need to invoke the `rgnMapFromActor()` service. Finally, it is not possible to use a semaphore in a memory region shared between a user actor and a supervisor actor.

- The above example would work in a similar fashion if the spawned actor did not impose the address of the created region, but used the `K_ANYWHERE` flag instead.
- The spawned actor maps only one page from the region created by the spawning actor, although this region is three pages long. Access to an address beyond the shared page would result in access to private data for the spawning actor, and in a memory fault for the spawned actor.
- The above example does not invoke the `rgnFree()` call. The region in the spawned actor will be freed at exit time. This does not mean that the physical memory will be freed as soon as the target actor disappears. Physical memory will be effectively freed when both actors have exited, regardless of the order in which they terminate.

Inter-actor Communication

This chapter explains how actors can communicate. It contains the following sections:

- “Introduction” on page 173 contains a summary of the communication methods available.
- “Message Queues” on page 174 explains how to use message spaces to communicate between actors.
- “Local Access Points” on page 184 explains how to use a local access point.
- “IPC” on page 187 explains how to use Inter-Process Communication (IPC).

Introduction

The ChorusOS operating system offers a set of services for communicating between actors. Sharing memory between two actors to enable them to communicate was described in “Sharing Memory Between Two Actors” on page 167. Other communication mechanisms can be split into two categories:

- Mechanisms which are said to be local: they do not enable 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 which may be transparently used in a distributed way. The IPC service enables actors to exchange messages in a transparent fashion whether they are running on the same machine or not.

Message Queues

This feature is designed to allow an application composed of one or multiple actors to create a shared communication environment, often referred to as message space, within which these actors can exchange messages efficiently. In particular, supervisor and user actors of the same application can use this feature to exchange messages. Furthermore, messages may be initially allocated and sent by interrupt handlers in order to be processed later by threads.

The feature is designed around the concept of message space which encapsulates within a single entity:

- a set of message pools shared by all actors of the application
- a set of message queues through which these actors exchange messages allocated from the shared message pools

A message space is a temporary resource which must be explicitly created by one actor within the application. Once created, a message space may be opened by other actors within the application. Actors which have opened the same message space are said to share this message space. A message space is automatically deleted when its creating actor and all actors which opened it have exited.

A message pool is defined by two parameters (message size and number of messages) provided by the application when it creates the message space. The configuration of the set of message pools defined within a message space depends upon the needs of the application.

A message is an array of bytes which can be structured and used at application level through any appropriate convention. Messages are presented to actors as pointers within their address space.

Messages are posted to message queues belonging to the same message space. All actors sharing a message space can allocate messages from the message pools. In the same way, all actors sharing a message space have send and receive rights on each queue of the message space.

Even though most applications only need to create a single message space, the feature is designed to allow an application to create or open multiple message spaces. However, messages allocated from one message space cannot be sent to a queue of a different message space. A typical use of message spaces is as follows:

1. The first actor, aware of the overall requirements of the application, creates the message space.
2. Other actors of the application open the shared message space.
3. An actor allocates a message from a message pool, and fills it with the data to be sent.

4. The actor which allocated the message can then post it to the appropriate queue, and can assign a priority to the message.
5. The destination actor can get the message from the queue. At this point, the message is removed from the queue.
6. Once the destination actor has processed the message, it may free the message so that the application may allocate it again. Alternatively, the destination actor could, for example, modify the message and post it again to another queue.

In order to make the service as efficient as possible, physical memory is allocated for all messages and data structures of the message space at message space creation. At message space open time, this memory is transparently mapped by the system into the actor address space. Further operations such as posting and receiving a message are done without any copy involved.

Creating a message space is performed as follows:

```
#include <mipc/chMipc.h>

int msgSpaceCreate (KnMsgSpaceId    spaceGid,
                   unsigned int     msgQueueNb,
                   unsigned int     msgPoolNb,
                   const KnMsgPool* msgPools);
```

The `spaceGid` parameter is a unique global identifier assigned by the application to the message space being created. This identifier is also used by other actors of the application to open the message space. Thus, this identifier serves to bind actors participating in the application to the same message space. The `K_PRIVATEID` predefined global identifier indicates that the message space created will be private to the invoking actor: its queues and message pools will only be accessible to threads executing within this actor. No other actor will be able to open that message space. The message space is described by the last three parameters:

- `msgQueueNb` indicates how many queues must be created within the message space. Each queue of the message space is then designated by its index within the set of queues. This may vary from 0 to `msgQueueNb - 1`.
- `msgPoolNb` is the number of message pools to be created in the message space.
- `msgPools` is a pointer to an array of `msgPoolNb` pool descriptions. Each pool is described by a `KnMsgPool` data structure which includes the following information:
 - `msgSize`, which defines the size of each message belonging to the pool
 - `msgNumber`, which defines how many messages of `msgSize` bytes must be created in this pool

Figure 8-1 shows an example of a message space recently created by an actor.

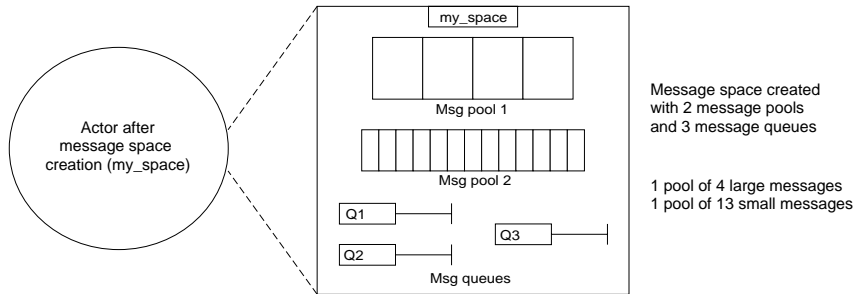


Figure 8-1 Creating a Message Space

The created message space is assigned a local identifier which is returned to the invoking actor as the return value of the `msgSpaceCreate()` call. The scope of this local identifier is the invoking actor.

A message space may be opened by other actors through the following call:

```
#include <mipc/chmipc.h>

int msgSpaceOpen(KnMsgSpaceId spaceGid);
```

The message space assigned with the `spaceGid` unique global identifier must have been previously created by a call to `msgSpaceCreate()`. A local identifier is returned to the invoking actor. This message space local identifier can then be used to manipulate messages and queues within the message space. Figure 8-2 shows an example of a message space recently opened by a second actor.

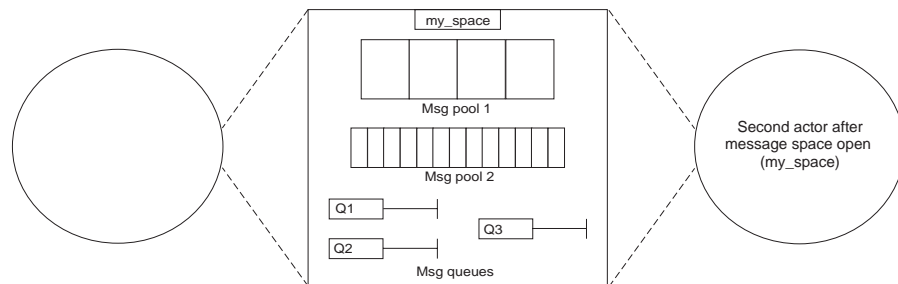


Figure 8-2 Opening a Message Space

A message may be allocated by the following call:

```
#include <mipc/chmipc.h>

int msgAllocate(int          spaceLid,
                unsigned int  poolIndex,
                unsigned int  msgSize,
                KnTimeVal*    waitLimit,
```



```
char**      msgAddr);
```

`msgAllocate()` attempts to allocate a message from the appropriate pool of the message space identified by the `spaceLid` return value of a previous call to `msgSpaceOpen()` or `msgSpaceCreate()`. If `poolIndex` is not set to `K_ANY_MSGPOOL`, the allocated message will be the first free (not yet allocated) message of the pool defined by `poolIndex`, regardless of the value specified by the `msgSize` parameter. Otherwise, if `poolIndex` is set to `K_ANY_MSGPOOL`, the message will be allocated from the first pool for which the message size fits the requested `msgSize`. In this context, first pool means the one with the lowest index in the set of pools defined at message space creation time. If the pool is empty, no attempt will be made to allocate a message from another pool.

If the message pool is empty (all messages have been allocated and none has been freed yet), `msgAllocate()` will block, waiting for a message in the pool to be freed. The invoking thread is blocked until the wait condition defined by `waitLimit` expires.

If successful, the address of the allocated message is stored at the location defined by `msgAddr`. The returned address is the address of the message within the address space of the actor. Remember that a message space is mapped within the address space of the actors sharing it. However, message spaces and, as a consequence, messages themselves, may be mapped at different addresses in different actors. This is specially true for message spaces shared between supervisor and user actors.

Figure 8–3 illustrates two actors allocating two messages from two different pools of the same message space.

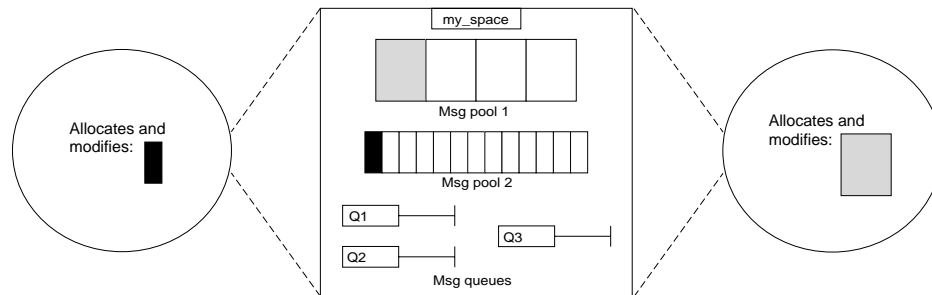


Figure 8–3 Allocating Messages from Pools

Once it has been allocated and initialized by the application, a message may be posted to a message queue with:

```
#include <mipc/chmipc.h>
```

```
int msgPut(int      spaceLid,
            unsigned int queueIndex,
            char*    msg,
            unsigned int prio);
```

`msgPut()` posts the message, the address of which is `msg`, to the message queue `queueIndex` within the message space, the local identifier of which is `spaceLid`. The message must have been previously allocated by a call to `msgAllocate()`. The message will be inserted into the queue according to its priority, `prio`. Messages with a high priority will be taken first from the queue.

Posting a message to a queue is done without any message copy, and may be done within an interrupt handler, or with preemption disabled.

Figure 8–4 illustrates the previous actors posting their messages to different queues.

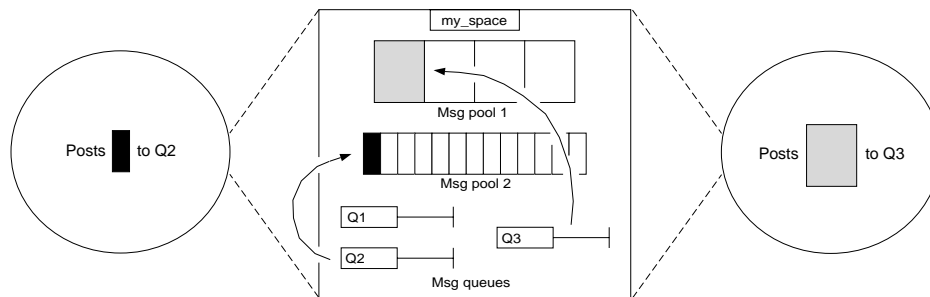


Figure 8–4 Posting Messages to Queues

Getting a message from a queue, if any, is achieved using:

```
#include <mipc/chmipc.h>
```

```
int msgGet(int          spaceLid,
            unsigned int queueIndex,
            KnTimeVal*   waitLimit,
            char**        msgAddr,
            KnUniqueId*   srcActor);
```

`msgGet()` enables the invoking thread to get the first message with the highest priority pending behind the message queue `queueIndex`, within the message space whose local identifier is `spaceLid`. Messages with equal priority are posted and delivered in a first-in first-out order.

The address of the message delivered to the invoking thread is returned at the location defined by the `msgAddr` parameter. If no message is pending, the invoking thread is blocked until a message is sent to the message queue, or until the time-out, if any, defined by the `waitLimit` parameter expires.

The `srcActor`, if non-null, points to a location where the unique identifier of the actor (referred to as the source actor) which posted the message is to be stored.

No data copy is performed to deliver the message to the invoking thread. Multiple threads can be blocked, waiting in the same message queue. At present it is not possible for one thread to wait for message arrival on multiple message queues. This

type of multiplexing mechanism could be implemented at the application level using the ChorusOS event flags facility.

Figure 8–5 illustrates previous actors receiving messages from queues.

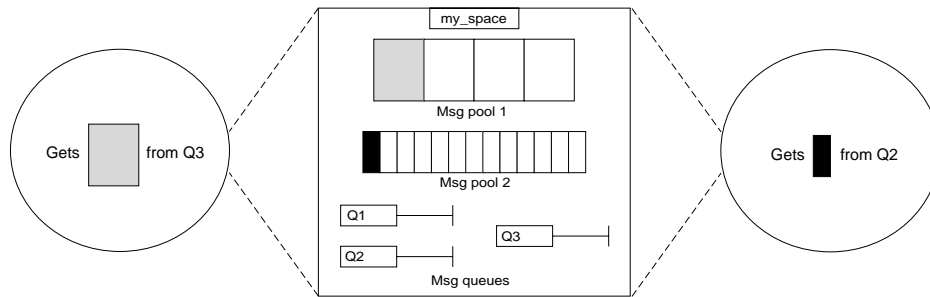


Figure 8–5 Getting Messages from Queues

A message which is of no further use to the application may be returned to its pool of messages available for further allocation with the following call:

```
#include <mipc/chMipc.h>

int msgFree(int      spaceLid,
            char*     msg);
```

Code Example 8–1 illustrates a very simple use of the message queue facility.

Refer to the `msgSpaceCreate(2K)`, `msgSpaceOpen(2K)`, `msgAllocate(2K)`, `msgPut(2K)`, `msgGet(2K)`, and `msgFree(2K)` man pages.

CODE EXAMPLE 8–1 Communicating Using Message Spaces

(file: `progov/msgSpace.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <chorus.h>
#include <mipc/chMipc.h>
#include <am/afexec.h>

AcParam param;

#define NB_MSG_POOLS 2
#define NB_MSG_QUEUES 3
#define SMALL_MSG_SZ 32
#define LARGE_MSG_SZ 256
#define NB_SMALL_MSG 13
#define NB_LARGE_MSG 4
#define SAMPLE_SPACE 1111
#define LARGE_POOL 0
#define SMALL_POOL 1
```

```

#define Q1 0
#define Q2 1
#define Q3 2

KnMsgPool samplePools[NB_MSG_POOLS];
char* tagPtr = "Spawned";

int main(int argc, char** argv, char**envp)
{
    int          res;
    int          msgSpaceLi;
    char*        smallMsg;
    char*        smallReply;
    char*        largeMsg;
    KnCap        spawnedCap;
    KnActorPrivilege actorP;

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
        printf("Cannot get actor privilege, error %d\n", res);
        exit(1);
    }

    if (argc == 1) {
        /*
         * This is the first actor (or spawning actor):
         *   Create a message space,
         *   Spawn another actor,
         *   Allocate, modify and post a small message on Q2
         *   Get a large Message from Q3, print its contents, free it
         *   Get reply of small message on Q1, print its contents, free it.
         */

        samplePools[LARGE_POOL].msgSize = LARGE_MSG_SZ;
        samplePools[LARGE_POOL].msgNumber = NB_LARGE_MSG;

        samplePools[SMALL_POOL].msgSize = SMALL_MSG_SZ;
        samplePools[SMALL_POOL].msgNumber = NB_SMALL_MSG;

        msgSpaceLi = msgSpaceCreate(SAMPLE_SPACE, NB_MSG_QUEUES,
                                    NB_MSG_POOLS, samplePools);
        if (msgSpaceLi < 0) {
            printf("Cannot create the message space error %d\n",
                  msgSpaceLi);
            exit(1);
        }

        /*
         * Message Space has been created, spawn the other actor,
         * argv[1] set to "Spawned" to differentiate the 2 actors.
         */
        param.acFlags = (actorP == K_SUPACTOR)? AFX_SUPERVISOR_SPACE :
                                                                AFX_USER_SPACE;
        res = afexeclp(argv[0], &spawnedCap, &param, argv[0], tagPtr,
                        NULL);
        if (res == -1) {
            printf("Cannot spawn second actor, error %d\n", errno);
            exit(1);
        }
    }
}

```

```

/*
 * Allocate a small message
 */
res = msgAllocate(msgSpaceLi, SMALL_POOL, SMALL_MSG_SZ,
    K_NOTIMEOUT, &smallMsg);
if (res != K_OK) {
    printf("Cannot allocate a small message, error %d\n", res);
    exit(1);
}

/*
 * Initialize the allocated message
 */
strncpy(smallMsg, "Sending a small message\n", SMALL_MSG_SZ);

/*
 * Post the allocated small message to Q2 with priority 2
 */
res = msgPut(msgSpaceLi, Q2, smallMsg, 2);
if (res != K_OK) {
    printf("Cannot post the small message to Q2, error %d\n", res);
    exit(1);
}

/*
 * Get a large message from Q3 and print its contents
 */
res = msgGet(msgSpaceLi, Q3, K_NOTIMEOUT, &largeMsg, NULL);
if (res != K_OK) {
    printf("Cannot get the large message from Q3, error %d\n",
        res);
    exit(1);
}

printf("Received large message contains:\n%s\n", largeMsg);

/*
 * Free the received large message
 */
res = msgFree(msgSpaceLi, largeMsg);
if (res != K_OK) {
    printf("Cannot free the large message, error %d\n", res);
    exit(1);
}

/*
 * Get the reply to small message from Q1 and print its contents
 */
res = msgGet(msgSpaceLi, Q1, K_NOTIMEOUT, &smallReply, NULL);
if (res != K_OK) {
    printf("Cannot get the small message reply from Q1, "
        "error %d\n", res);
    exit(1);
}

printf("Received small reply contains:\n%s\n", smallReply);

/*
 * Free the received small reply
 */

```

```

res = msgFree(msgSpaceLi, smallReply);
if (res != K_OK) {
    printf("Cannot free the small reply message, error %d\n", res);
    exit(1);
}

} else {

    /*
     * This is the spawned actor:
     * Check we have effectively been spawned
     * Open the message space
     * Allocate, initialize and post a large message to Q3
     * Get a small message from Q2, print its contents
     * Modify it and repost it to Q1
     */

    int    1;

    if ((argc != 2) || (strcmp(argv[1], tagPtr) != 0)) {
        printf("%s does not take any argument!\n", argv[0]);
        exit(1);
    }
    /*
     * Open the message space, using the same global identifier
     */
    msgSpaceLi = msgSpaceOpen(SAMPLE_SPACE);
    if (msgSpaceLi < 0) {
        printf("Cannot open the message space error %d\n",
            msgSpaceLi);
        exit(1);
    }

    /*
     * Allocate the large message
     */
    res = msgAllocate(msgSpaceLi, K_ANY_MSGPOOL, LARGE_MSG_SZ,
        K_NOTIMEOUT, &largeMsg);
    if (res != K_OK) {
        printf("Cannot allocate a large message, error %d\n", res);
        exit(1);
    }

    strcpy(largeMsg, "Sending a very large large large large large message\n");

    /*
     * Post the large message to Q3 with priority 0
     */
    res = msgPut(msgSpaceLi, Q3, largeMsg, 0);
    if (res != K_OK) {
        printf("Cannot post the large message to Q3, error %d\n", res);
        exit(1);
    }

    /*
     * Get the small message from Q2
     */
    res = msgGet(msgSpaceLi, Q2, K_NOTIMEOUT, &smallMsg, NULL);
    if (res != K_OK) {

```

```

        printf("Cannot get the small message from Q2, error %d\n", res);
        exit(1);
    }

    printf("Spawned actor received small message containing:\n%s\n", smallMsg);

    for (l = 0; l < strlen(smallMsg); l++) {
        if ((smallMsg[l] >= 'a') && (smallMsg[l] <= 'z')) {
            smallMsg[l] = smallMsg[l] - 'a' + 'A';
        }
    }

    /*
     * Post the small message back to Q1, with priority 4
     */
    res = msgPut(msgSpaceLi, Q1, smallMsg, 4);
    if (res != K_OK) {
        printf("Cannot post the small message reply to Q1, error %d\n",
            res);
        exit(1);
    }
}
return 0;
}

```

- Two actors are used, one spawned by the other. The first actor:
 - creates a message space with two pools of messages and three message queues as shown in the previous figure
 - allocates a small message, initializes it and posts it to a queue
 - waits for a large message on a second queue, prints its contents and deallocates it
 - waits for the small message to come back on a third queue, prints its contents, deallocates it, and terminates
- Meanwhile, the second actor:
 - opens the message space, allocates a large message to be initialized and sends it to the first actor
 - receives the small message, converts all lower case characters to upper case, and posts it back to the third queue before terminating

Local Access Points

Local Access Points (LAPs) are a low overhead mechanism for calling service routines in supervisor actors on the local site by both user and supervisor actor calls.

The following is an example of the use of local access points. Refer to the `lapInvoke(2K)`, `svLapBind(2K)`, `svLapCreate(2K)`, and `svLapDelete(2K)` man pages.

CODE EXAMPLE 8-2 Creating and Invoking LAPs

```
(file: progov/lap.c)

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <chorus.h>
#include <lap/chLap.h>
#include <am/afexec.h>
#include <exec/chExec.h>

AcParam param;

KnCap      actorCap;
KnLapDesc  lapDesc;
char*      lapArgument;
KnTimeVal  timeval;

void
lapHandler(char* message, char* cookie)
{
    int      res;

    res = actorSelf(&actorCap);
    if (res != K_OK) {
        printf("actorSelf failed, returns %d\n",res);
        exit(1);
    }

    printf("LAP handler is running \n");
    printf("    thread LI = %d, actor UI = 0x%x 0x%x\n",
           threadSelf(), actorCap.ui.uiHead, actorCap.ui.uiTail);
    printf("    Argument = %s, cookie = %s\n",message, cookie);
}

int main(int argc, char** argv, char** envp)
{
    int      res;
    KnCap     clientCap;
    KnActorPrivilege actorP;
    char*     cookie = "Chorus";

    res = actorPrivilege(K_MYACTOR, &actorP, NULL);
    if (res != K_OK) {
```



```

        printf("Cannot get actor privilege, error %d\n", res);
        exit(1);
    }

    if (actorP != K_SUPACTOR) {
        printf("This program must be run in supervisor mode\n");
        exit(1);
    }

    if (argc == 1) {
        printf("Must be run with one argument: LAP name\n");
        exit(1);
    }

    if (argc == 2) {
        /*
         * This is the Server actor.
         * connect a LAP handler,
         * bind a symbolic name to this LAP (name given as argv[1])
         * spawn a client actor (give the LAP symbolic name in argument)
         * wait one minute for Lap invocation
         */

        /*
         * Spawn the client actor
         */
        param.acFlags = AFX_SUPERVISOR_SPACE;
        res = afexecvp(argv[0], &clientCap, &param, argv[0], argv[1],
            "ARGH", NULL);
        if (res == -1) {
            printf("Cannot spawn client actor, error %d\n", errno);
            exit(1);
        }

        /*
         * Create the LAP
         */
        res = svLapCreate(K_MYACTOR, (KnLapHdl) lapHandler,
            cookie, K_LAP_SETJMP, &lapDesc);
        if (res != K_OK) {
            printf("svLapCreate failed, returns %d\n", res);
            exit(1);
        }

        /*
         * Bind a symbolic name
         */
        res = svLapBind(&lapDesc, argv[1], 0);
        if (res != K_OK) {
            printf("svLapBind failed, returns %d\n", res);
            exit(1);
        }

        /*
         * Wait one minute
         * Other client actors can be run from the console:
         * rsh target arun lap.r lap-name lap-argument
         */
        timeval.tmSec = 60;
    }

```

```

timeval.tmNSec = 0;
threadDelay(&timeval);

/*
 * Unbind the LAP name and Delete the LAP
 */

res = svLapUnbind(argv[1]);
if (res != K_OK) {
    printf("svLapUnBind failed, returns %d\n",res);
    exit(1);
}

res = svLapDelete(&lapDesc);
if (res != K_OK) {
    printf("svLapDelete failed, returns %d\n",res);
    exit(1);
}
printf("Server actor is leaving ...\n");
} else {

/*
 * This is the Client Actor:
 * argv[1] is the LAP name, argv[2] is the LAP argument.
 * get the LAP descriptor and invoke the LAP handler.
 */

res = actorSelf(&actorCap);
if (res != K_OK) {
    printf("actorSelf failed ! Return code = %d\n",res);
    exit(1);
}

printf("Client actor is running, thread li = %d, "
       "actor UI = 0x%x 0x%x\n",
       threadSelf(), actorCap.ui.uiHead, actorCap.ui.uiTail);

/*
 * Get the LAP descriptor knowing its name
 */
res = lapResolve(&lapDesc, argv[1], 0);
if (res != K_OK) {
    printf("lapResolve failed, returns %d\n", res);
    exit(1);
}

/*
 * Invoke the LAP handler
 */
res = lapInvoke(&lapDesc, argv[2]);
if (res != K_OK) {
    printf("lapInvoke failed, returns %d\n", res);
    exit(1);
}

printf("Client actor is leaving ...\n");
}
return 0;
}

```

- The main thread:
 - checks if it is running as a supervisor actor
 - spawns another copy of itself using `afexec()`
 - creates a Local Access Point and connects a LAP handler which prints the unique identifier of the current thread (Actor UI + thread LI), the LAP argument and the LAP cookie on the console
 - binds a symbolic name received as the first argument
 - waits for one minute for LAP invocations
 - frees the LAP and its name, then terminates the program
- The spawned actor:
 - receives two arguments: the symbolic LAP name and the argument to be passed to the LAP handler
 - retrieves the LAP descriptor
 - invokes the LAP handler, then terminates

IPC

IPC is a set of programming interfaces that allow you to create and manage individual program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with other users. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods. IPC methods include:

- Pipes and named pipes
- Message queueing
- Semaphores
- Shared memory
- Sockets

The following is an example of the use of the IPC mechanisms provided in the ChorusOS operating system.

Refer to the `grpAllocate(2K)`, `grpPortInsert(2K)`, `ipcReceive(2K)`, `ipcSend(2K)`, `ipcTarget(2K)`, `portCreate(2K)`, and `portDelete(2K)` man pages.

CODE EXAMPLE 8-3 Communicating Using IPC

```
(file: progov/ipcSend.c)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <chorus.h>
#include <am/afexec.h>
#include "ipc/chIpc.h"

#define ABORT_DELAY 1000                /* Delay for ipcReceive */

static KnUniqueId    thePortUi;        /* Our port unique identifier */
static int           thePortLi;        /* ..... local identifier */
static KnCap         groupCap;         /* Our group capability */
static KnCap         clientCap;        /* Capability of the spawned */
/* actor */

/* The outgoing annex and body */
static char sndAnnex[] = "Hello world from Chorus ...\n";
static char sndBody[] = "The sea is calm, the tide is full ...\n";

/* The received annex and body */
static char rcvAnnex[K_MSGANNEXSIZE];
static char rcvBody[1000];

/* Parameter for actor spawning */
AcParam param;

/* Port group stamp */
char* stamp;
#define MYSTAMP 100
#define STAMPSIZE 10

int main(int argc, char** argv, char** envp)
{
    int          rslt;                /* Work */
    KnMsgDesc    smsg;                /* Descriptor for message being sent */
    KnMsgDesc    rmsg;                /* Descriptor for message being received */
    KnIpcDest    ipcDest;             /* IPC address */

    if (argc == 1) {
        /*
         * Server actor
         * Create the destination port
         */
        thePortLi = portCreate(K_MYACTOR, &thePortUi);
        if (thePortLi < 0) {
            printf("portCreate failed, returns %d\n", thePortLi);
            exit(1);
        }
        /*
         * Allocate a port group and insert the port into it

```

```

        */
    rslt = grpAllocate(K_STATUSER, &groupCap, MYSTAMP);
    if (rslt < 0) {
        printf("grpAllocate failed, returns %d\n", rslt);
        exit(1);
    }

    rslt = grpPortInsert(&groupCap, &thePortUi);
    if (rslt < 0) {
        printf("grpPortInsert failed, returns %d\n", rslt);
        exit(1);
    }

    /*
     * Spawn the client actor
     * The group stamp is given in argument
     */
    stamp = malloc(STAMPSIZE);
    sprintf(stamp, "%d", MYSTAMP);

    rslt = afexecldp(argv[0], &clientCap, NULL, argv[0], stamp, NULL);
    if (rslt == -1) {
        printf("Cannot spawn client actor, error %d\n", errno);
        exit(1);
    }

    /*
     * Receive the message
     */
    rmsg.flags = 0;
    rmsg.bodySize = sizeof(rcvBody);
    rmsg.bodyAddr = (VmAddr)rcvBody;
    rmsg.annexAddr = (VmAddr)rcvAnnex;

    rslt = ipcReceive(&rmsg, &thePortLi, ABORT_DELAY);
    if (rslt < 0) {
        printf("ipcReceive failed, returns %d\n", rslt);
        exit(1);
    }

    printf ("%s\n%s\n", rcvAnnex, rcvBody);

    rslt = portDelete(K_MYACTOR, thePortLi);
    if (rslt < 0) {
        printf("portDelete failed, returns %d\n", rslt);
        exit(1);
    }
} else {
    /*
     * Get the port group capability giving the stamp.
     * Stamp has been received in argv[1]
     */
    rslt = grpAllocate(K_STATUSER, &groupCap, (int) atoi(argv[1]));
    if (rslt < 0) {
        printf("grpAllocate failed, returns %d\n", rslt);
        exit(1);
    }

    /*
     * Prepare the message descriptor for the message to send
     */
    smsg.flags = 0;
    smsg.bodySize = sizeof(sndBody);

```

```

    smsg.bodyAddr = (VmAddr)sndBody;
    smsg.annexAddr = (VmAddr)sndAnnex;
    /*
     * Prepare the IPC address for the message destination.
     * Send the message in broadcast mode.
     */
    ipcDest.target = groupCap.ui;
    rslt = ipcTarget(&ipcDest.target, K_BROADMODE);
    if (rslt < 0) {
        printf("ipcTarget failed, returns %d\n", rslt);
        exit(1);
    }

    /* Send from our DEFAULT port */
    rslt = ipcSend(&smsg, K_DEFAULTPORT, &ipcDest);
    if (rslt < 0) {
        printf("ipcSend failed, returns %d\n", rslt);
        exit(1);
    }
}

return 0;
}

```

■ The main thread:

- creates a port
- creates a port group and inserts the port into it
- spawns another copy of itself, using `afexec()`, and passes the port group stamp as an argument
- waits for a message on the created port
- prints the contents of the body and the annex
- frees any used resources and terminates

■ The spawned actor:

- retrieves the port group capability passing the stamp received as an argument
- prepares and sends a message to this group in broadcast mode (annex and body of the message are initialized with strings)
- terminates

Time Management

The ChorusOS operating system offers five time management services:

- Tick service
- Date service
- Time-out service
- Timer service
- Virtual time and virtual time-out service

The configuration of your ChorusOS operating system determines which services are available.

Time Management Services

The following time management services are available:

- The `tick` service enables the system to manage the clock, counting ticks since the boot of the system. Thus the only time available is the time elapsed since the last reboot.
- The `date` service enables the ChorusOS operating system to maintain a current date, usually expressed in seconds since the 01/01/1970. Calls to `set` and `get` the time of day are available, through standard C libraries `ctime` and `localtime`, and are not detailed in this document.
- The `time-out` service enables supervisor actors to set up time-outs. A time-out may be roughly described as a callback which will be performed when a given delay has expired. Callbacks are performed using a special invocation mechanism (called Local Access Point or LAP) reserved for supervisor actors.

- The `timer` service is an extension of the time-out mechanism, enabling user and supervisor actors to set up call backs in a more flexible fashion.
- The virtual time and time-out service allows you to measure the CPU time used by threads, and to define handlers which will be called if a per-thread or per-actor CPU quota is reached.

Table 9-1 shows which services are available for a given configuration:

TABLE 9-1 Time Management Service Availability

Service	Availability
tick	always available
date	configured with <code>DATE</code>
time-out	always available
timer	configured with <code>TIMER</code>
virtual time	configured with <code>VTIMER</code>

Current Time

An actor, whether user or supervisor, may get the time elapsed since the last reboot through the following system call:

```
#include <exec/chTime.h>

int sysTime(KnTimeVal* time);
```

This will fill in the time data structure which is built from two fields:

- `tmSec` which indicates the number of whole seconds elapsed since the last reboot
- `tmNSec` which indicates the number of nanoseconds

The resolution of the value depends on the platform on which the system is running, and may be obtained by a call to:

```
#include <chorus.h>

int sysTimeGetRes(KnTimeVal* resolution);
```


The time value returned at the location defined by the resolution parameter represents the smallest possible difference between two distinct values of the system time.

Timers

This feature provides timer services for both user and supervisor actors. One-shot as well as periodic timers are provided. Time-out notification is achieved through user-provided handler threads which are woken up in the application actor.

The timer facility uses the concept of a timer object within the actor. These timer objects may be created, deleted and set dynamically. Once created, they are addressed by local identifiers within the context of the actor and are deleted automatically when the actor terminates.

The application is expected to create one or more threads dedicated to timer notification handling, by declaring themselves ready to handle these types of events. The relationship between a timer object and a thread (or a set of threads) is established through a `threadPool` object which is used to block threads waiting for the expiration of a timer.

Thus, the basic mechanism for dealing with timers is:

1. Allocate and initialize a `threadPool` object.
2. Create one thread which will block on the `threadPool` object.
3. Create a timer associated with the above `threadPool` object.
4. Set the timer (effectively arm it).

The second and third steps may take place in any order. When timer expiration occurs, the dedicated thread will be unblocked so that it may now perform any operation which should be done due to timer expiration. For example, it may print a warning message, re-arm the timer (unless it was a periodic timer), and block itself again. As usual with the ChorusOS operating system data structures, these `threadPool` objects must be pre-allocated by the application.

A `threadPool` object is initialized as follows:

```
#include <etimer/chEtimer.h>

int timerThreadPoolInit(KnThreadPool* threadPool);
```

A timer may then be created as follows:

```
#include <etimer/chEtimer.h>

int timerCreate(KnCap*      actorCap,
               int         clockType,
               KnThreadPool* threadPool,
               void*       cookie,
               int*        timerLi);
```

This creates a timer object in the actor defined by the `actorCap` parameter. Applications will usually use `K_MYACTOR`. When the timer is armed and reaches expiration, one of the threads blocked on the `threadPool` object will be selected and awakened. This thread will be passed the `cookie` parameter of the `timerCreate()` call. When successful, `timerCreate()` returns the local identifier of the created timer at the location defined by the `timerLi` parameter. The only clock type currently supported is `K_CLOCK_REALTIME`, and corresponds to the time returned by `sysTime()`.

A thread may block itself on a `threadPool` object through the following call:

```
#include <etimer/chEtimer.h>

int timerThreadPoolWait(KnThreadPool* threadPool,
                      void**       cookie,
                      int*         overrun,
                      KnTimeVal*   waitLimit);
```

The `threadPool` object must have been previously initialized. `timerThreadPoolWait()` blocks the invoking thread until a timer associated with `threadPool` expires or until the `waitLimit` condition is reached. Upon timer expiration, the thread will return from this call, and the `cookie` field will have been updated with the value associated with the timer.

The `overrun` counter is used to indicate to the thread that either the time-out notification has been delayed (in this case the `overrun` value is 1) or that a number of time-out notifications have been lost (in this case the `overrun` value is strictly greater than 1).

A timer may be armed with:

```
#include <etimer/chEtimer.h>

int timerSet(KnCap*      actorCap,
            int         timerLi,
            int         flag,
            KnITimer*   new,
            KnITimer*   old);
```

This call arms the timer defined by the first two parameters where `timerLi` is the timer identifier as returned by `timerCreate()`. `timerSet()` allows the

specification of the time-out using either a relative or an absolute time using the `flag` parameter. The time-out is specified using the new parameter which is a structure containing the following fields:

- `KnTimeVal ITmValue`. This field specifies at what time the time-out will occur for the first (and maybe only) time.
 - If the flag is set to `K_TIMER_ABSOLUTE`, the time value is an absolute time (in terms of time as managed by the `sysTime` service).
 - If the flag is set to `K_TIMER_INTERVAL`, the time value is a delay relative to the current time.
- `KnTimeVal ITmReload`. This field contains the subsequent interval for a periodic timer. If its value is 0, the timer will be a one-shot timer.

If the `old` parameter is non-null, the time remaining before timer expiration is returned at the location defined by `old`. If `new` is non-null and the timer has already been set, the current setting is cancelled and replaced with the new one. If the new time specified is 0, the current setting will simply be cancelled. If `new` is set to null, the current setting specification is left unchanged.

Refer to the `timerThreadPoolInit(2K)`, `timerCreate(2K)`, `timerSet(2K)`, `timerThreadPoolWait(2K)`, and `sysTime(2K)` man pages.

The following example illustrates the use of timer services for both user and supervisor actors.

CODE EXAMPLE 9-1 Using Timers

```
(file: progov/timers.c)

#include <stdio.h>
#include <stdlib.h>
#include <chorus.h>
#include <etimer/chEtimer.h>

KnThreadPool samplePool;
int           periodic;
int           oneShot;
int           periodicLid;
int           oneShotLid;

#define USER_STACK_SIZE (1024 * sizeof(long))

KnSem  sampleSem; /* Semaphore allocated as global variable */

int
childCreate(KnPc entry)
{
    KnActorPrivilege  actorP;
    KnDefaultStartInfo_f startInfo;
    char*             userStack;
    int                childLid = -1;

```

```

int                res;

startInfo.dsType      = K_DEFAULT_START_INFO;
startInfo.dsStackSize = K_DEFAULT_STACK_SIZE;

res = actorPrivilege(K_MYACTOR, &actorP, NULL);
if (res != K_OK) {
    printf("Cannot get the privilege of the actor, error %d\n", res);
    exit(1);
}

if (actorP == K_SUPACTOR) {
    startInfo.dsPrivilege = K_SUPTHREAD;
} else {
    startInfo.dsPrivilege = K_USERTHREAD;
}

if (actorP != K_SUPACTOR) {
    userStack = malloc(USER_STACK_SIZE);
    if (userStack == NULL) {
        printf("Cannot allocate user stack\n");
        exit(1);
    }
    startInfo.dsUserStackPointer = userStack + USER_STACK_SIZE;
}

startInfo.dsEntry = entry;

res = threadCreate(K_MYACTOR, &childLid, K_ACTIVE, 0, &startInfo);
if (res != K_OK) {
    printf("Cannot create the thread, error %d\n", res);
    exit(1);
}

return childLid;
}

void
timerWait(int myThLi)
{
    /* do nothing */
}

void
sampleThread()
{
    int      myThLi;
    int      res;
    void*    cookie;
    int      overrun;
    KnITimer periodicTimer;
    KnTimeVal tv;

    myThLi = threadSelf();
    printf("Thread %d started\n", myThLi);

    for(;;) {
        res = timerThreadPoolWait(&samplePool, &cookie, &overrun, K_NOTIMEOUT);
        if (res != K_OK) {
            printf("Cannot wait on thread pool, error %d\n", res);

```

```

        exit(1);
    }
    if (overrun != 0) {
        printf("Thread %d. We were late! overrun set to : %d\n",
            myThLi, overrun);
    }
    if (cookie == &periodic) {
        printf("Thread %d. Time is flying away!\n", myThLi);
    } else if (cookie == &oneShot) {
        printf("Thread %d. Isn't it time to go home?\n", myThLi);
        periodicTimer.ITmValue.tmSec = 0; /* seconds */
        periodicTimer.ITmValue.tmNSec = 0; /* nanoseconds */
        periodicTimer.ITmReload.tmSec = 0; /* seconds */
        periodicTimer.ITmReload.tmNSec = 0; /* nanoseconds */
        res = timerSet(K_MYACTOR, periodicLid, NULL, &periodicTimer, NULL);
        if (res != K_OK) {
            printf("Cannot cancel periodic timer, error %d\n", res);
            exit(1);
        }
        /*
         * Periodic timer is cancelled
         * Get current time,
         * Wait for a short while (3 seconds) and quit
         */
        res = sysTime(&tv);
        if (res != K_OK) {
            printf("Cannot get system time, error %d\n", res);
            exit(1);
        }
        printf("Current system time is %d seconds\n", tv.tmSec);
        printf("No more periodic messages should be printed now!\n");
        K_MILLI_TO_TIMEVAL(&tv, 3000);
        threadDelay(&tv);
        /* We are all done ! */
        exit(0);
    } else {
        printf("Spurious timer!\n");
    }
} /* for() */
}

int main(int argc, char** argv, char** envp)
{
    int res;
    KnTimeVal tv;
    int thLi1;
    int thLi2;
    KnITimer periodicTimer;
    KnITimer oneShotTimer;

    res = timerThreadPoolInit(&samplePool);
    if (res != K_OK) {
        printf("Cannot initialize thread pool, error %d\n", res);
        exit(1);
    }

    res = timerCreate(K_MYACTOR, K_CLOCK_REALTIME, &samplePool,
        &periodic, &periodicLid);
    if (res != K_OK) {
        printf("Cannot create periodic timer, error %d\n", res);
    }
}

```

```

        exit(1);
    }

    res = timerCreate(K_MYACTOR, K_CLOCK_REALTIME, &samplePool,
        &oneShot, &oneShotLid);
    if (res != K_OK) {
        printf("Cannot create one shot timer, error %d\n", res);
        exit(1);
    }

    thLi1 = childCreate((KnPc)sampleThread);
    thLi2 = childCreate((KnPc)sampleThread);

    res = sysTime(&tv);
    if (res != K_OK) {
        printf("Cannot get system time, error %d\n", res);
        exit(1);
    }
    printf("Current system time is %d seconds\n", tv.tmSec);

    periodicTimer.ITmValue.tmSec = 1; /* seconds */
    periodicTimer.ITmValue.tmNSec = 0; /* nanoseconds */
    periodicTimer.ITmReload.tmSec = 1; /* seconds */
    periodicTimer.ITmReload.tmNSec = 0; /* nanoseconds */
    res = timerSet(K_MYACTOR, periodicLid, NULL, &periodicTimer, NULL);
    if (res != K_OK) {
        printf("Cannot arm periodic timer, error %d\n", res);
        exit(1);
    }

    oneShotTimer.ITmValue.tmSec = tv.tmSec + 30; /* seconds */
    oneShotTimer.ITmValue.tmNSec = 0; /* nanoseconds */
    oneShotTimer.ITmReload.tmSec = 0; /* seconds */
    oneShotTimer.ITmReload.tmNSec = 0; /* nanoseconds */

    res = timerSet(K_MYACTOR, oneShotLid, K_TIMER_ABSOLUTE,
        &oneShotTimer, NULL);
    if (res != K_OK) {
        printf("Cannot arm one shot timer, error %d\n", res);
        exit(1);
    }

    res = threadDelete(K_MYACTOR, K_MYSELF);
    if (res != K_OK) {
        printf("Cannot suicide myself, error %d\n", res);
        exit(1);
    }

    return 0;
}

```

- The main thread sets up everything that is needed, so that two created threads will respond to a single periodic timer of one second for a duration of thirty seconds.
- The thirty-second period is bounded by a one-shot timer handled by the same pool of two threads.

- Before starting, the current system time is printed.
- When the thirty second timer has elapsed, the periodic timer is cancelled and the current system time is printed again.
- A small delay has been added before the actor terminates to check that the periodic timer has been cancelled correctly.

PART **IV** Debugging and Performance Profiling

System and Application Debugging

This chapter presents the source-level debugging architecture in the ChorusOS 4.0 operating system. It explains how to configure the different servers and tools, and how to use them.

- “Preparing the System for Symbolic Debugging” on page 203 describes how to prepare your system for symbolic debugging.
- “Application Debugging Architecture” on page 207 gives an overview of the application debugging architecture and lists the steps involved in setting up a system debugging session.
- “System Debugging Architecture” on page 210 gives an overview of the system debugging architecture and lists the steps involved in setting up an application debugging session.
- “Sample XRAY Start-up Script” on page 223 presents a script which can be used to start the XRAY Debugger for the ChorusOS operating system.

Preparing the System for Symbolic Debugging

Compiling for Debugging

In order to use all the debugging features in the XRAY Debugger, you need to generate symbolic debugging information when you compile components. This information is stored in the object files and describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

How you enable debugging in components will depend on which release of ChorusOS 4.0 you have. The binary release of ChorusOS 4.0 includes the source code for the BSP, driver and example components. These you will compile in what is known as an *imake build environment* because the `imake` tool is used to create the Makefiles for these components.

The source release of ChorusOS 4.0 includes everything in the binary release plus source code for system components, such as the kernel and the operating system. These components are built in an *mkmk build environment* where the tool `mkmk` is used to build Makefiles. For more details see *ChorusOS 4.0 Production Guide*.

Enabling Debugging for Components Built with `imake`

To build all your components with symbolic debugging information turned on:

- Edit the `Paths` file located in the root of your build directory, created after you run the `configure` command, and add the following line to the end:

```
FREMOTEDEB=ON
```

Other ways can be used to selectively build your components with symbolic debugging information. These are presented below.

To enable symbolic debugging throughout the component directory and its sub-directories:

1. Edit the `Project.tpl` file located in the root of the component source directory, and add the following line to the end:

```
DEBUG=$(DEBUG_ON)
```

2. Change directory to the root of your build directory and remove all the object files and executables:

```
% make clean
```

3. Rebuild the local Makefile:

```
% make Makefile
```

4. Rebuild the sub-directory Makefiles:

```
% make Makefiles
```

5. Finally, rebuild the component:

```
% make
```

To enable symbolic debugging in selected component directories:

1. Edit the `Imakefile` within each desired component source directory, and add the following line to the end:

```
DEBUG=$(DEBUG_ON)
```

2. Change directory to the root of your build directory and remove all the object files and executables:

```
% make clean
```

3. Rebuild the local Makefile:

```
% make Makefile
```

4. Finally, rebuild the component:

```
% make
```

If you prefer not to modify the `Imakefile` or `Project.tmpl` files, there is an alternative way of enabling debugging. You can pass the debug option within the `make` command itself:

- Change to your build directory and remove all the object files and executables:

```
% make clean
```

Rebuild the component with symbolic debugging enabled:

```
% make MAKE="make DEBUG=-gdwarf-2"
```

- You can also create a `DEBUG` environment variable. If you use the C shell:

```
% setenv DEBUG -gdwarf-2
```

If you use the Bourne shell:

```
$ DEBUG=-gdwarf-2  
$ export DEBUG
```

Now call `make` with the `-e` option to import environment variables:

```
% make -e
```

Once a component has been compiled in debug mode, rebuild and reboot the system image.

Enabling Debugging for Components Built with mkmk

To enable symbolic debugging for system components:

1. Change to your build directory and remove all the object files and executables:

```
% make clean
```

2. Create a mkmk build definition file:

```
% echo 'FREMOTEDEB=ON' > filename.df
```

filename can be a name of your choice.

3. Rebuild the system component:

```
% make makemk
```

Configuring the Debug Agent

The DebugAgent is activated by enabling the `DEBUG_SYSTEM` feature with the `configurator(1CC)` command:

```
% configurator -set DEBUG_SYSTEM=true
```

Note - The `DEBUG_SYSTEM` feature is set to `true` by default.

When the DebugAgent is activated, communications on the serial line are performed in binary mode.

The DebugAgent has eight tunable options that you can configure with `ews` or `configurator`. The following three tunables control the behavior of the DebugAgent when it is enabled (`DEBUG_SYSTEM=true`):

- `dbg.agent.startup` specifies the behavior of the DebugAgent when the DebugServer attempts to connect to the target. Possible values are `stop` or `resume`. In `stop` mode the system will wait indefinitely until the DebugServer connects to the DebugAgent. In `resume` mode the system will wait for no more than one second. In both cases, the DebugAgent will issue the prompt `DebugAgent: trying to sync with DebugServer... over the serial line and`

sound a beep. If the DebugServer does not connect, the system will continue booting and console operations will be performed raw, without any processing by the DebugAgent, on the serial line.

- `dbg.agent.exceptionmode` specifies the behavior of the DebugAgent when an exception is raised before the DebugServer has connected. Possible values are `catch` or `forward`. In `catch` mode the DebugAgent catches all exceptions and blocks the target until a DebugServer has connected and resumed the execution of the target. In `forward` mode the DebugAgent forwards all exceptions directly to the kernel-installed handlers.
- `dbg.agent.consolemode` specifies the operating mode of the system console. Possible values are `sync` or `async`. In `sync` mode the target is blocked until each message has been transmitted to the host for output and acknowledged. In `async` mode, console output is buffered and transmitted to the host periodically, either when the buffer is full or on each timer interrupt, whichever occurs first.

The following five tunables control the serial line used by the DebugAgent.

- `dbg.agent.device` specifies the serial device used by the DebugAgent. Possible values are `COM1`, `COM2`, `COM3`, or `COM4`. `COM1` refers to communication port 1 (COM 1) on a PC, or the first serial line on other boards. `COM2` refers to communication port 2 (COM 2) on a PC, or the second serial line on other boards, and so on.
- `dbg.agent.baud` specifies the baud rate of the serial line. Possible values are: 115200, 57600, 38400, 19200, 9600, 4800, 2400, or 1200.
- `dbg.agent.parity` specifies the parity of the serial line. Possible values are `none`, `even`, or `odd`.
- `dbg.agent.databits` specifies the number of data bits. The only possible value is 8.
- `dbg.agent.stopbits` specifies the number of stop bits. Possible values are 1 or 2.

Note - When the DebugAgent is not active (`DEBUG_SYSTEM=false`), the serial line is used by the system debugging console, and the five tunables control the serial device and speed.

Application Debugging Architecture

This section describes the components within the application debugging architecture.

Architecture Overview

The application debugging architecture has two components:

- XRAY Debugger for ChorusOS
- Remote debug server (RDBC) `rdbc(1CC)`

The XRAY Debugger for ChorusOS runs on the host. The remote debug server runs on the target and communicates with XRAY over the Ethernet. This is illustrated in Figure 10-1.

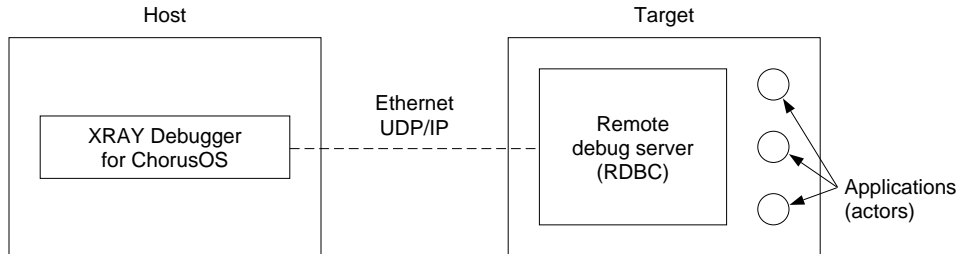


Figure 10-1 Application Debugging Architecture

Application debugging is intended to be used for debugging user applications, dynamically loaded actors, as well as certain supervisor actors. It is not possible to debug the Actor Management (AM) or I/O Management components (IOM), the kernel, or the system drivers. Application debugging relies on the RDBC supervisor actor which uses the services of the AM, the IOM, the kernel, and system drivers such as the Ethernet driver. When an application is debugged, only that application is affected. Other applications in the operating system, as well as the operating system itself, will keep running.

Setting up a Debugging Session

To begin an application debugging session, follow these steps:

1. Ensure that your target is connected to your network.
2. Prepare the system for symbolic debugging. See “Preparing the System for Symbolic Debugging” on page 203 for information on how to do this.
3. Configure and start `rdbc`, the ChorusOS remote debug server. See “RDBC Configuration and Usage” on page 209 and `rdbc(1CC)`.
4. Configure and start the XRAY Debugger for ChorusOS. See “Sample XRAY Start-up Script” on page 223.

RDBC Configuration and Usage

The RDBC server can be started automatically or manually:

- For RDBC to be started automatically, the `conf/sysadm.ini` file, read during system initialization by `C_INIT`, must contain a command which mounts the NFS root. If this `mount` command is present, edit the `conf/sysadm.ini` file and add the following line after it:

```
rdbc
```

Note - It is extremely important that RDBC is started *after* the NFS root is mounted on the target.

- RDBC can be started manually, as a normal application, as follows:

```
% rsh -n name arun rdbc
```

Note - Running RDBC manually gives you freedom to choose when you want to carry out application debugging, freeing up valuable resources.

To stop RDBC, use the `akill` command. First identify the actor process ID (*aid*):

```
% rsh name aps
```

Then kill the RDBC process:

```
% rsh name akill aid
```

Note - Your XRAY application debug session will be lost if you stop RDBC.

Information about what targets are available to XRAY is held in the file `chorusos.brd`. There are four columns: the machine names where RDBC executes are specified in the first column, slot numbers are specified in the second column, and the last two columns are for comments. XRAY interprets integer values between 0 and 25 in the second column as slot numbers and larger values as TCP/IP port numbers, and will adapt its connection to the server accordingly. The default TCP/IP port number of RDBC is 2072.

Here is an example `chorusos.brd` file:

```
target-i386  2072  "i386"  "Application debug of target-i386"
target-ppc   2072  "ppc"   "Application debug of target-ppc"
```

The entries specify the application debug of actors running on `target-i386` and `target-ppc` respectively, and require that RDBC be running on both machines.

Two or more RDBC servers can be run on the same target to provide you with a separate console for each program being debugged.

See `rdbc(1CC)` for more information.

Note - The name and port number specified in `chorusos.brd` have different meanings:

- For application debug, the `chorusos.brd` file specifies the target name, and the port number corresponds to a UDP/IP port number, 2072 by default.
 - For system debug, the `chorusos.brd` file specifies the host name where the RDBS server is running. The port number corresponds to the RDBS slot number in the range 0..25.
-

System Debugging Architecture

This section describes the components within the system debugging architecture.

Architecture Overview

The system debugging architecture has the following components:

- XRAY Debugger for ChorusOS
- Remote debug server (RDBS) `rdbs(1CC)`
- ChorusOS debug server (DebugServer) `chserver(1CC)`
- Debug agent (DebugAgent)

The first three components run on the host. The fourth, the debug agent, runs on the target and communicates with the ChorusOS debug server through a serial connection. This is illustrated in Figure 10-2.

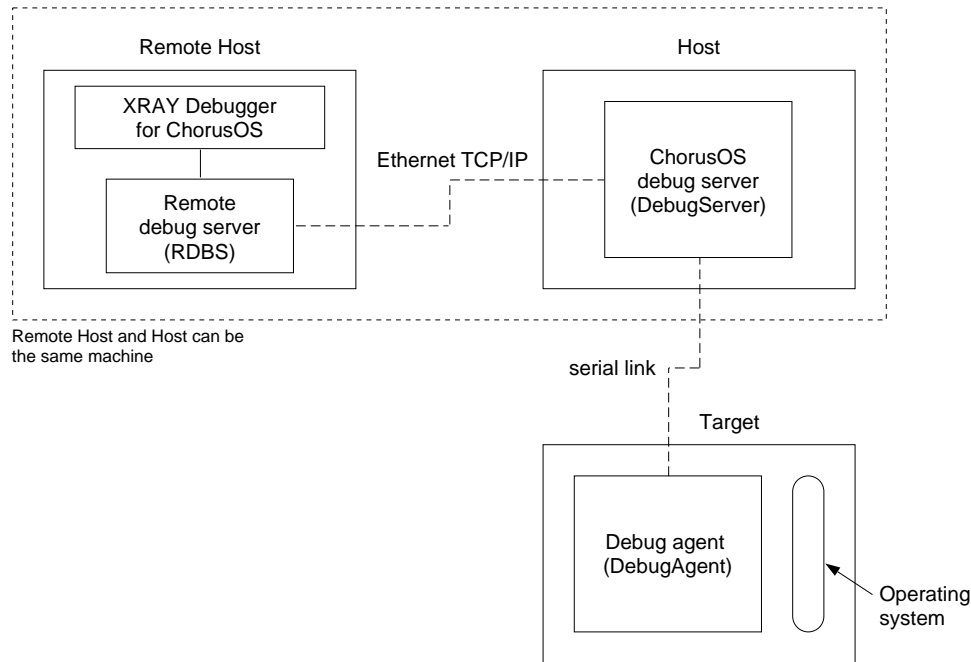


Figure 10-2 System Debugging Architecture

A more detailed description of the debugging architecture can be found in Chapter 2 and 3 of the *ChorusOS Debug Architecture and API Specifications* document (</opt/SUNWconn/SEW/4.0/chorus-doc/pdf/DebugApi.pdf>).

System debugging is intended to be used for debugging different parts of the ChorusOS operating system. This includes the kernel, the system drivers, the BSP, and those supervisor actors that cannot be debugged with application debugging such as the AM, and the IOM. System debugging also allows you to debug interrupt and exception handlers. During system debugging, the whole operating system is affected.

Setting up a Debugging Session

To begin your first system debugging session, follow these steps:

1. Connect a serial cable between the host and target.
2. Prepare the system for symbolic debugging. See “Preparing the System for Symbolic Debugging” on page 203 for information on how to do this.
3. Start and configure the ChorusOS DebugServer `chserver`. See “Setting up a Debugging Session” on page 211 and `chserver(1CC)`.
4. Register the target with `chadmin`, the ChorusOS DebugServer administration tool. See “Registering a Target” on page 214.

5. Configure and start `rdbs`, the ChorusOS debug server for the XRAY Debugger. See “DebugServer Configuration File” on page 213 and `rdbs(1CC)`.
6. Start the ChorusOS debug console `chconsole`. See the `chconsole(1CC)` man page for further details.
7. Configure and start the XRAY Debugger for ChorusOS. See “Sample XRAY Start-up Script” on page 223.

Note - If you do not start `chserver` you will not be able to use `chconsole`. However, you can still view the system console using the `tip` or `cu` commands. See `tip(1)` and `cu(1C)` for more details.

For subsequent debugging sessions, you need only perform the following steps:

1. Start the ChorusOS DebugServer `chserver`, if it is not already running.
2. Start `rdbs`, the ChorusOS debug server for the XRAY debugger, if it is not already running.
3. Start the ChorusOS debug console `chconsole`.
4. Start the XRAY Debugger for ChorusOS.

Starting and Configuring the ChorusOS DebugServer

Identifying the Serial Device

The ChorusOS DebugServer `chserver` communicates with the target through a serial cable connection and must be run on the host to which the target is connected.

To identify the serial device, look in the `/etc/remote` file. This file contains references to remote systems that you can access through your local serial line. For more details, see `remote(4)`. The device is usually named `/dev/ttya` or `/dev/ttyb` and will be the same device used by the `tip` or `cu` tools. The device must be readable and writable by all users.

DebugServer Slot Numbers

The DebugServer is a Sun RPC server that is registered with the `rpcbind` server. When you require more than one debug server to run on the same host, assign a unique slot number (in the range 0..65535) to each of them so that individual debug

servers can be identified. If only one debug server is started on a given host, it is not necessary to allocate a slot number as 0 will be used by default.

If you decide to assign a slot number to your DebugServer, use the DebugServer environment variable CHSERVER_HOST.

DebugServer Environment Variable

The DebugServer, as well as all the other tools based on the Debug Library, uses the optional environment variable CHSERVER_HOST. This environment variable indicates:

- the host name where your DebugServer is running
- optionally, the slot number for the DebugServer RPC service

The format of the environment string is `host[:slot-id]`. For example:

```
% setenv CHSERVER_HOST jeriko
% setenv CHSERVER_HOST concerto:3
```

DebugServer Configuration File

Configuration information about targets is held in a special file which the DebugServer reads every time you run it. For each target, the configuration file contains:

- The name of the target
- The serial device used to communicate with the target
- Configuration parameters for the serial device (baud rate, parity)
- The architecture type of the target (i386, PPC, SPARC)
- The absolute path of the `layout.xml` image layout file generated by `mkimage`

When a new target is registered, see “Registering a Target” on page 214 for details of how to do this, the configuration file is modified.

Starting the DebugServer

On the host to which your target or targets are connected, type the following command:

```
% chserver
```

This will start the DebugServer as a background process. An empty configuration file called `dbg_config.xml` is copied into your home directory the first time you run the DebugServer.

If you have defined a slot number *n* and not set the environment variable, you can start the DebugServer as follows:

```
% chserver -slot n
```

A complete description of the DebugServer is given in the `chserver(1CC)` man page.

Note - If `chserver` is run on a different host to the one the system image was built on, particularly on a shared file system with a different view of the build directory, the tool will not be able to access the necessary source files during system debugging. This problem is NFS related, the symbolic link created on one host may not be valid for another, and is due to there being relative file references in the `layout.xml` file. There are two solutions to the problem:

- Break the symbolic link with the file
 `/build-NUCLEUS/conf/mkimage/layout_typedef.xml` accessed by
 `chserver`.
 - Copy the `conf` and `image` directories to the host where `chserver` will run, then use `chadmin` to set the path of the `layout.xml` file.
-

Stopping the DebugServer

Stop the DebugServer by using the `chadmin` tool.

```
% chadmin -shutdown
```

Registering a Target

Before registering a target you need to know:

- The name of the target
- The name of the serial device
- The path of the `layout.xml` file generated by `mkimage` (the path of this file is printed by `mkimage` when a system image is built)

Now you can register the target by typing:

```
% chadmin -add-serial-target name/  
-device device/  
-layout-file layout_file
```

name is the name of your target, *device* is the serial device that you have identified, and *layout_file* is the absolute path of the `layout.xml` file.

You only need to register a target once as configuration information is saved in your `dbg_config.xml` file.

Updating Target Information

Use `chadmin` to update the information that you gave during the registration of your target.

The following example sets the baud rate to 38400, the parity to none, and uses the device `/dev/ttya` for the target *name*:

```
% chadmin -baud 38400 -parity none -device /dev/ttya name
```

If you wish to specify a new `layout.xml` file because the path has changed (see note), use the following command to inform the DebugServer of the new path:

```
% chadmin -layout-file path/layout.xml name
```

Note - When you change the mode of system image booting (using the `BOOT_MODE` global variable) or select a different system image configuration (using the `SYSTEM` global variable), the path to the `layout.xml` file will change.

For example, if you type:

```
% configurator -set SYSTEM=kts
```

The path will change to `build_dir/image/RAM/kts/layout.xml`.

If you then type:

```
% configurator -set SYSTEM=chorus
```

The path will change to `build_dir/image/RAM/chorus/layout.xml`.

Similarly, if you change the mode of system image booting:

```
% configurator -set BOOT_MODE=ROM
```

The path will change to `build_dir/image/ROM/chorus/layout.xml`.

Deactivating a Target

A target can be deactivated to disconnect the DebugServer from the DebugAgent and release the serial device used by the DebugServer. When a target is deactivated, the DebugAgent switches to a stand-alone mode. The `chconsole` must no longer be used as the DebugServer does not read the serial line any more. Instead, you must start the `tip(1)` or `cu(1C)` tools to gain access to the system debugging console.

A target may be temporarily removed (deactivated) with the following command:

```
% chadmin -deactivate name
```


name is the name of your target.

When a target is deactivated, it is not removed from the DebugServer configuration file so that it is possible to reactivate it later.

Reactivating a Target

Before the target can be reactivated, you must stop any `tip` or `cu` tools which may be using the serial line.

The target is reactivated with this command:

```
% chadmin -activate name
```

name is the name of your target.

The DebugServer will synchronize with the DebugAgent and the DebugAgent will switch to a binary protocol mode. At this stage, you must use the `chconsole` to gain access to the system debugging console.

Removing a Target

A target may be permanently removed by first deactivating it (see “Deactivating a Target” on page 216) then unregistering it with this command:

```
% chadmin -remove-target name
```

name is the name of your target.

Providing you have deactivated the target first, the target’s configuration information will be deleted from the configuration file.

RDBS Configuration and Usage

If RDBS is started without any parameters it will connect, by default, to the first target available on the DebugServer. However, you can specify a target name on the command-line provided the name you use is registered with the DebugServer.

Note - This name is unrelated to the name under which the target might be known on the TCP/IP network (through another connection). It only identifies the serial line connecting the target with the DebugServer.

A complete set of command-line parameters are documented in `rdbshost(1CC)`.

Several RDBS servers may be run on one machine to debug several targets, provided you define a different slot for each server.

Information about what targets are available to XRAY is held in the file `chorusos.brd`. There are four columns: the machine names where RDBS executes are specified in the first column, slot numbers are specified in the second column, and the last two columns are for comments. XRAY interprets integer values between 0 and 25 in the second column as slot numbers and larger values as TCP/IP port numbers, and will adapt its connection to the server accordingly.

Here is an example `chorusos.brd` file:

```
rdbshost      0      "i386"  "System debug of target-i386"
rdbshost      1      "ppc"    "System debug of target-ppc"
```

The entries specify that two copies of RDBS will run on the `rdbshost` machine (a Solaris workstation): one on slot 0, configured to debug the `target-i386` target, and another on slot 1, configured to debug the `target-ppc` target.

Note - The name and port number specified in `chorusos.brd` have different meanings:

- For application debug, the `chorusos.brd` file specifies the target name, and the port number corresponds to a UDP/IP port number, 2072 by default.
 - For system debug, the `chorusos.brd` file specifies the host name where the RDBS server is running. The port number corresponds to the RDBS slot number in the range 0..25.
-

Concurrent System and Application Debugging

Combine the example `chorusos.brd` files given in “RDBC Configuration and Usage” on page 209 and “RDBS Configuration and Usage” on page 217:

```
rdbshost      0      "i386"  "System debug of target-i386"
rdbshost      1      "ppc"    "System debug of target-ppc"
target-i386   2072   "i386"  "Application debug of target-i386"
target-ppc    2072   "ppc"    "Application debug of target-ppc"
```

The first two entries specify that two copies of RDBS will run on the `rdbshost` machine (a Solaris workstation): one on slot 0, configured to debug the `target-i386` target, and another on slot 1, configured to debug the `target-ppc` target.

The last two entries specify the application debug of actors running on `target-i386` and `target-ppc` respectively, and require that RDBC be running on both machines.

By attaching to the first and third targets, you can carry out application and system debugging on the same target concurrently. However, while the system is stopped it is not possible to carry out application debug because halting the system halts RDBC, as well as the application itself.

Example XRAY/RDBS debug session

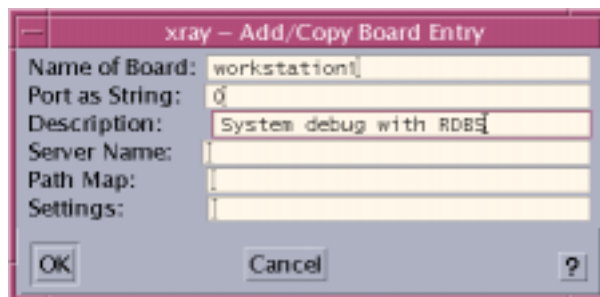
In this session the target is named `target-i386`, the workstation is named `workstation1` and all host tools are available. Several actors and drivers have been added to the system, and they have been compiled for system debugging.

Make sure you have enabled the system debugging during system generation (see “Compiling for Debugging” on page 203), then run `DebugServer` (see “Starting the `DebugServer`” on page 213) and connect a console to `target-i386`. Run RDBS in the following manner:

```
% rdbbs target-i386
```

Run XRAY (by using the “Sample XRAY Start-up Script” on page 223, for example), then go to the `Managers` window and select the `Connect` tab.

Select the `Boards->Add or Copy board` entry to register your target for system debug. XRAY opens the `Add/Copy Board Entry` pop-up dialog:



Enter the host name where RDBS is running in the `Name of Board` field. Enter the slot number used by RDBS (0 by default) in the `Port as String` field. Leave the other fields blank.

Note - On Windows NT, XRAY uses native Windows pathnames and it not aware of the Cygwin UNIX emulation layer used by the ChorusOS host tools. As a result, pathname translations must be specified so that XRAY can translate the Unix-like pathnames returned by the DebugServer, or embedded in object modules, into native Windows NT pathnames. Typical pathname translations are `/c/=C:\` and `/d/=D:\`. They must reflect the results of the Cygwin `mount` command.

After the dialog box is validated, the new board appears in the window. Connect to the RDBS server by double-clicking on it with the left mouse button.

This will connect you to RDBS, and through it to the DebugServer and the target. You can now view the system as it runs.

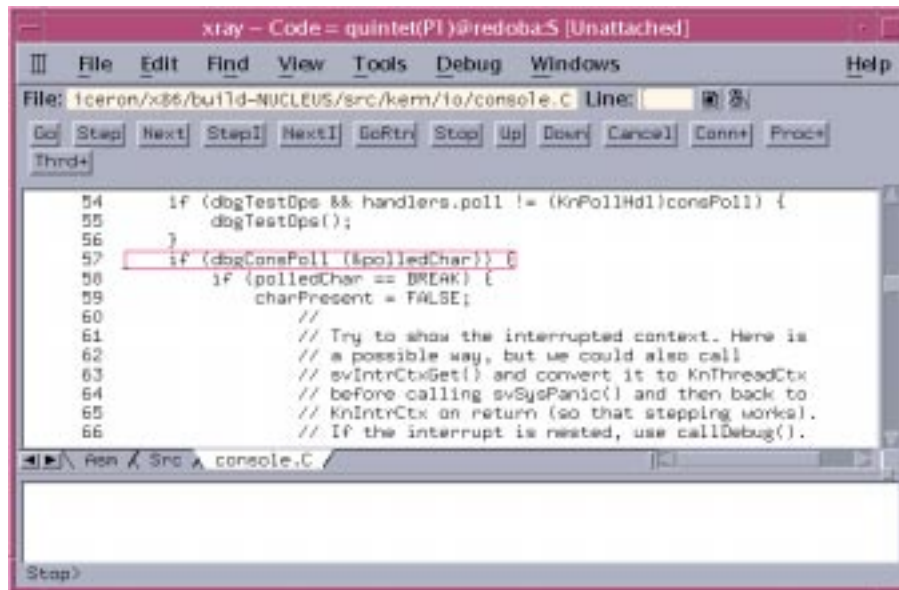
Enter the following in the command-line area of the `Code` window to see a list of actors running on the target:

```
Conn> stat actors
```

Select the `Process` tab in the `Managers` window. The `Available Processes` list will show a single entry representing the system as process number 1. Double-clicking on it will stop the system and initiate a debugging session.

XRAY will present you with the list of actors for which symbols should be loaded. By default, all actors are selected and you can press the `OK` button. XRAY will find the executable files automatically. For some of them, it may not have the path and it will prompt for the pathname of the missing executable file. If the actor's binary file is statically linked, you must indicate the path where it is located. If the actor's binary file is relocatable, then your only option is to select `Cancel`, because system debugging does not support the debugging of actors loaded from relocatable binaries.

After all selected actors have been loaded, XRAY will show where the system has been stopped in the `Code` window. The name of the thread which was executing last, also known as the current thread, will be displayed in the title bar. Thread execution can now be controlled.



Think of a function you want to debug, `myFunc()` for example, and perform this command:

```
% scope myFunc
```

XRAY displays the source code of the function in the `Code` window. You can place a breakpoint in it for the current thread by double-clicking with the left button on the selected line number. This will set a per-thread breakpoint, for the current thread.

If you do not know whether the current thread will execute this function, place a global breakpoint by opening a local menu with the right mouse button and selecting `Set Break All Threads`. Press the `Go` button to resume running the function.

Once the breakpoint has been reached, examine the values of variables by double-clicking on them with the right mouse button.

If the breakpoint is not reached, and the system continues to run, you can stop it asynchronously using the `Stop` button. The `Code` window will show the stop location.

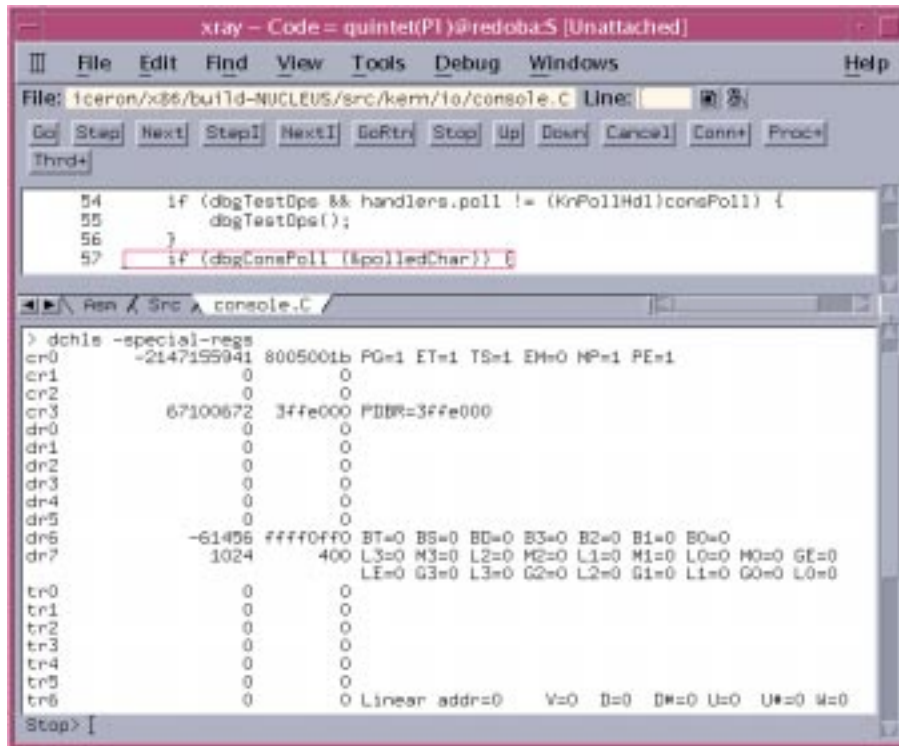
Note - Due to the way in which the stop operation has been implemented, this will always be the same location inside the clock interrupt handler, except if the system was blocked in a console input, or performing console output.

You can find the interrupted location by examining the stack with the `Up` button.

The chls tool

The `chls` tool is available from the XRAY command window with the `dchls` command. You can use this command to display values which are not directly visible in the XRAY windows. For example, to look at the processor specific registers, type the following command:

```
Stop> dchls -special-regs
```



Troubleshooting

If the DebugServer process is terminated, RDBS will attempt to reconnect to a new DebugServer process automatically. If there was a debugging session open at the time, the single process representing the ChorusOS operating system will be killed, and the debugging context lost. You will need to re-grab the process after RDBS has reconnected to the new DebugServer. If this does not work, then kill and restart RDBS.

If the target is rebooted, the single process representing the ChorusOS operating system will appear in the XRAY output window first as killed, and shortly after as restarted. XRAY will then attempt to reinsert all previously set breakpoints and promote them from thread-specific to global. Any breakpoints that cannot be reinserted will be deleted.

If you stop the system while it is waiting for console input, it will not resume until you provide some keyboard input.

Currently, the DebugServer does not offer access to the target's ChorusOS IPC ports. RDBS will report this by printing a warning message on start-up.

If a given symbol is present in several actors, or in several modules in a single actor (a static symbol, for example), then you can use the `ps /f symbol_name` command to display all of the occurrences of the symbol, complete with a full pathname. The full pathname is of the `@binary_file\module\symbol_name` form. Use this full pathname to reference symbols which not in the current scope.

Because XRAY asks for a thread list each time a debugged process stops, and generating the list takes a long time during system debugging, the thread list shown in the `Threads Manager` is simplified. It does not include fields names, such as actor names, and is only updated when the current thread changes. The full thread list is available from the `Resource Viewer` or through the `dallthreads` command. Per-actor threads can be displayed with the command `dthreads=aid`. You can force the full thread list to be displayed, both in the `Resource Viewer` and in the `Threads Manager`, by permanently leaving the `Resource Viewer` window open on the thread list.

Sample XRAY Start-up Script

A sample script for setting up and starting XRAY is provided below.

Create a sub-directory within your system image directory to hold the script. For example, if your system image is called `chorus.RAM`, the directory would be `image/RAM/chorus/bin`.

Remember to modify the line which initializes the `XRAY_INSTALL_DIR` environment variable to point to the directory where XRAY is installed. This directory also contains the `bin`, `docs`, `docschxx`, `license`, `master`, `xraycore` and `xrayrdb` sub-directories. The script assumes you have put the `license.dat` file into this directory.

This shell script works if you use either a time-limited licence for XRAY, or a license locked to your machine. Please refer to the XRAY documentation for details of the other options available.

```
#!/bin/sh
set +x
```

```

XRAY_INSTALL_DIR=<xray_install_dir>

# Clean up possible crash
/bin/rm -f core /tmp/.MasterMsg/.MasterSock.$DISPLAY*

# Prepare environment variables
XRAYMASTER=$XRAY_INSTALL_DIR/master
export XRAYMASTER

USR_MRI=$XRAY_INSTALL_DIR
export USR_MRI

LD_LIBRARY_PATH=$XRAYMASTER/lib
export LD_LIBRARY_PATH

# LM_LICENSE_FILE=$XRAY_INSTALL_DIR/license.dat
LM_LICENSE_FILE=/Work/build/mir/mri/mri/license.dat
export LM_LICENSE_FILE

# If you use a license server, the following line starts it
# ./mri/bin/lmgrd
# Then we change the LM_LICENSE_FILE variable to point to the server
# LM_LICENSE_FILE=port_number@machine_name

# Run XRAY itself
$XRAY_INSTALL_DIR/master/bin/xray -VABS=rdb $*
# ./mri/master/bin/xray $*

```


Performance Profiling

This chapter explains how to analyze the performance of a ChorusOS operating system and its applications by generating a performance profile report.

- “Introduction to Performance Profiling” on page 225 explains why a performance profile is useful and how it can be used.
- “Preparing to Create a Performance Profile” on page 227 explains how to configure your system so that you can generate a performance profile.
- “Running a Performance Profiling Session” on page 228 explains how to create a performance profile.
- “Analyzing Performance Profiling Reports” on page 229 explains how to analyze the performance profile.
- “Performance Profiler Description” on page 231 gives more information on how the performance profiler works.

Introduction to Performance Profiling

The ChorusOS operating system performance profiling system contains a set of tools that facilitate the analysis and optimization of the performance of the ChorusOS operating system and applications. These tools concern only system components sharing the system address space, that is, the ChorusOS operating system components and supervisor application actors. This set of tools is composed of a profiling server, libraries for building profiled actors, a target controller and a host utility.

Software performance profiling consists of collecting data about the dynamic behavior of the software, to gain knowledge of the time distribution within the software. For

example, the performance profiling system is able to report the time spent within each procedure, as well as providing a dynamically constructed call graph.

The typical steps of an optimization project are:

1. To bench a set of typical applications, using the ChorusOS operating system and applications at peak performance. The selection of these applications is critical, as the system will eventually be tuned for this type of application.
2. To evaluate and record the output of the benchmarks.
3. To use the performance profiling system to collect raw data about the dynamic behavior of the applications.
4. To generate, evaluate and record the performance profiling reports.
5. To plan and implement optimizations such as rewriting certain time-critical routines in assembly language, using in-line functions or tuning algorithms.

The performance profiling tools provide two different classes of service, depending on the way in which the software being measured has been prepared:

- The performance profiling system is applied to software generated in the standard way, (the same version as used for benchmarking). In this case, the performance profiling reports only minimal information, which consists mainly of the percentage of time spent in each routine of the software. The corresponding performance profiling report is called *simple form*.
- The performance profiling system is applied to software regenerated exclusively for performance profiling; software is completely recompiled, using the performance profiling C compiler option (usually the `-p` option). This allows the performance profiling system to report much more information by dynamically counting routine invocations and building a complete call graph. The corresponding performance profiling report is called *full form*.

Note - The standard (binary) version of the ChorusOS operating system is not compiled with the performance profiling option: profiling the system will only generate a simple form. Non-profiled components (or components for which a simple report form is sufficient) do not need to be compiled with the performance profiling option.

In order to obtain a full form for ChorusOS operating system components, a source product distribution is needed. In this case, it is necessary to regenerate the system components with the performance profiling option set.

Preparing to Create a Performance Profile

Configuring the System

In order to perform system performance profiling using the ChorusOS Profiler, a ChorusOS target system must include the `ACTOR_EXTENDED_MNGT` and `NFS_CLIENT` feature options.

Launch the performance profiling server (the `PROF` actor) dynamically, using:

```
% rsh -n target arun PROF &
```

Compiling the Application

If you require full report forms, the profiled components must be compiled using the performance profiling compiler options (usually, the `-p` option).

If you are using the `imake` environment provided with the ChorusOS operating system, you can set the profiling option in the `Project.tmpl` file if you want to profile the whole project hierarchy, or in each `Imakefile` of the directories that you want to profile if you want to profile only a subset of your project hierarchy. In either case, add the following line:

```
PROF=$( PROF_ON)
```

You can also add the performance profiling option dynamically by calling `make` with the compiler profiling option:

```
% make PROF=-p
```

in the directory of the program that is to be performance profiled.

Launching the Performance Profiled Application

In this section, it is assumed that the application consists of a single supervisor actor, `the_actor`, it is also assumed that the target system is named `trumpet`, and that the target tree is mounted under the `$CHORUS_ROOT` host directory.

In order to be performance profiled, an application may be either:

- launched at system boot time, as part of the system image, or
- dynamically launched using the `arun` service, using the `-k` option, with the following command:

```
% rsh trumpet arun -k "the_actor"
```

Running a Performance Profiling Session

Starting the Performance Profiling Session

Performance profiling is initiated by running the `profctl` utility on the target system, using the `-start` option. This utility (see “Security” on page 64 for more details) considers the components to be profiled as arguments.

If *the_actor* was part of the system image:

```
% rsh trumpet arun profctl -start -b the_actor
```

Otherwise, if *the_actor* was loaded dynamically:

```
% rsh trumpet arun profctl -start -a the_actor aid
```

where *aid* is the numeric identifier of the actor (as returned by the `arun` or `aps` commands).

Note - Several components may be specified to the `profctl` utility. See “Security” on page 64 for more details.

Run the application.

Stopping the Performance Profiling Session

Performance profiling is stopped by running the `profctl` utility again, using the `-stop` option:

```
% rsh trumpet arun profctl -stop
```

When performance profiling is stopped, a raw data file is generated for each profiled component within the `/tmp` directory of the target file system. The name of the file consists of the component name, to which the suffix `.prof` is added. For example, if only `the_actor` was profiled, the file `$CHORUSUS_ROOT/tmp/the_actor.prof` would be created.

Generating Performance Profiling Reports

Performance profiling reports are generated by the `profrpg` host utility (see “Security” on page 64 for details on reporting options).

Use the report generator to produce a report for each profiled component; as follows:

```
% cd $CHORUSUS_ROOT/tmp
```

```
% profrpg the_actor > the_actor.rpg
```

In order to track the benefits of optimization, the reports should be archived.

Analyzing Performance Profiling Reports

Performance profiling is applied to a user-selected set of components (ChorusOS operating system kernel, supervisor actors). The result of the performance profiling consists of a set of reports, one per profiled component.

A performance profiling report consists of two parts:

- A global report that provides general information about the profiling session, including clock attributes, CPU attributes, and the distribution of CPU time between idle threads, user actors, non-profiled supervisor components, and each of the profiled supervisor components.
- A component-based function table that indicates the distribution of CPU time inside the profiled component.

For each function, the performance profile report displays the following information:

- **Function header.**
 - **Function number.** This field indicates the function number in the current report. It is provided in order to facilitate study of the report using a text editor.
 - **Function name.** This field indicates the name of the function.
 - **Size.** This field indicates the size of the function in bytes.
 - **Time spent in function.** This field indicates the flat time spent in the body of the function (the number of profiling ticks that occurred while an instruction was executed within the function). This value is followed by the percentage of the total component time it represents. This is the most valuable information and the report can be sorted by this key if desired.
 - **Total time spent in function.** This field indicates the aggregated-time spent within the function and called functions. The value is given as a percentage of total actor time. By default, the report generator sorts the table by this key. This field is computed by the report generator, and assumes that each call to a given routine lasts the same amount of time. This information is only provided in the full profiling form. In the simple form, this information is the same as the flat-time information.
 - **Recursion indicator.** This field indicates that the procedure was found in a recursive loop. As the profiling system is not fully set up for multithreading, this indicator might be erroneously set. This information is only provided in the full profiling form.
- **Call graph description.**
 - **List of callers.** This field details a list of the functions calling the profiled function. For each caller, the report provides:
 - the caller's function number
 - the number of calls
 - the caller's name and the offset of the call in the caller's body. When a function calls another function from several locations, several entries are made in the list of callers
 - **List of called functions.** For each called function, the report provides:
 - the callee's function number
 - the number of calls
 - the percentage of the total function time that is charged to the callee
 - the name of the function

Shown below is an example of a profiling report.

```

overhead=2.468
memcpy 4 K=18.834
memcpy 16 K=51.936
memcpy 64 K=185.579
memcpy 256 K=801.300
sysTime=2.576
threadSelf=2.210
thread switch=5.777
threadCreate (active)=8.062
threadCreate (active, preempt)=10.071
threadPriority (self)=3.789
threadPriority (self, high)=3.195
threadResume (preempt)=6.999
threadResume (awake)=4.014
...
ipcCall (null, timeout)=35.732
ipcSend (null, funcmode)=7.723
ipcCall (null, funcmode)=31.762
ipcSend (null, funcumode)=7.924
ipcCall (null, funcumode)=31.864
ipcSend (annex)=8.294
ipcReceive (annex)=7.086
ipcCall (annex)=33.708
ipcSend (body 4b)=8.020
ipcReceive (body 4b)=6.822
ipcCall (body 4b)=32.558
ipcSend (annex, body 4b)=8.684
ipcReceive (annex, body 4b)=7.495
ipcCall (annex, body 4b)=34.849

```

Performance Profiler Description

This section provides information about the performance profiling system's design, to help you understand the sequence of events that occurs before the generation of a performance profiling report.

The performance profiling tool set consists of:

- The profiler server, `PROF`, a supervisor actor. This actor first interprets performance profiling requests issued by the `PROF` utility, and then executes the performance profiling function at a selected profiling clock rate on the target. See `PROF(1CC)` for more details.
- The `profctl` target utility (see `profctl(1CC)`). This utility sends performance profiling requests to the profiler server, `PROF`, on the target.
- The `profrpg` (see `profrpg(1CC)`) host utility. This command interprets profiling data and produces coherent profiling reports on the development host.

The Performance Profiling Library

When the performance profiling compiler option (generally `-p`) is used, the compiler provides each function entry point with a call to a routine, normally called `mcount`. For each function, the compiler also sets up a static counter, and passes the address of this counter to `mcount`. The counter is initialized at zero.

What is done by `mcount` is defined by the application. Low-end performance profilers simply count the number of times the routine is called. ChorusOS Profiler provides a sophisticated `mcount` routine within the profiled library that constructs the runtime call graph. Note that you can supply your own `mcount` routine, for example to assert predicates when debugging a component.

The Performance Profiler Server

The profiler server, `PROF`, is a supervisor actor that can locate and modify static data within the memory context of the profiled actors, using the embedded symbol tables. The profiler server also dynamically creates and deletes the memory regions that are used to construct the call graph and count the profiling ticks (see below).

The Performance Profiling Clock

While the performance profiler is active, the system is regularly interrupted by the profiling clock, which by default is the system clock. At each clock tick, the instruction pointer is sampled, the active procedure is located and a counter associated with the interrupted procedure is incremented. A high rate performance profiling clock could use a significant amount of system time, which could lead to the system appearing to run more slowly. A rapid sampling clock could jeopardize the system's real-time requirements.

Notes About Accuracy

Significant disruptions in the real-time capabilities of the profiled programs must be expected, because performance profiling is performed by software (rather than by hardware with an external bus analyzer or equivalent device). Performance profiling using software slows down the processor, and the profiled applications may behave differently when being profiled compared to when running at full processor speed.

When profiling, the processor can spend more than fifty percent of the processing time profiling clock interrupts. Similarly, the time spent recording the call graph is significant, and tends to bias the profiling results in a non-linear manner.

The accuracy of the reported percentage of time spent is about five percent when the number of profiling ticks is in the order of magnitude of ten times the number of bytes in the profiled programs. In other words, in order to profile a program of 1 million bytes with any degree of accuracy, at least 10 millions ticks should be used. This level of accuracy is usually sufficient to plan code optimizations, which is the primary goal of the profiler, but the operator should beware of using all the fractional digits of the reported figures.

If more accuracy is needed, the operator can experiment with different combinations of the rate of the profiling clock, the type of profiling clock and the time spent profiling.

Configuring IPC

This appendix describes how to configure the IPC feature within the ChorusOS operating system to provide either local IPC communication, remote IPC communication over Ethernet, or remote IPC communication over the VME bus.

- “Generic IPC Configuration” on page 235 introduces the three different IPC configurations, and how to add them to the ChorusOS operating system.
- “Specific IPC Configuration” on page 237 describes remote IPC configuration in more detail.

Generic IPC Configuration

IPC Feature Configuration

The ChorusOS IPC feature is an optional component of the ChorusOS kernel which can be added in three different configurations:

1. local IPC. This configuration provides only local IPC communications.
2. local IPC + remote IPC. This configuration enables local and remote IPC communications to take place over a network data-link such as Ethernet or ATM (Asynchronous Transfer Method). This data-link is also called an external data-link, which means that the data-link driver is implemented within an independent driver outside of the kernel. It is also unreliable.
3. local IPC + remote IPC over the VME bus. This configuration allows local and remote IPC communications to take place over the VME bus.

The command-line configuration tool `configurator(ICC)` is used to set up each configuration.

To configure the local IPC feature:

```
% configurator -set IPC=true
```

To configure the local IPC + remote IPC feature:

```
% configurator -set IPC_REMOTE=true  
% configurator -set IPC_REMOTE_COMM=EXT
```

To configure the local IPC + remote IPC feature over the VME bus:

```
% configurator -set IPC_REMOTE=true  
% configurator -set IPC_REMOTE_COMM=VME
```

Site Number Administration

The IPC feature has the concept of a *site number*, a 32-bit unsigned integer which uniquely identifies a target board. Applications exchange messages through IPC ports, which are designated by a global identifier which includes the site number of the target board where the port is located.

The site number of a target is sent to the kernel at boot time in one of two ways:

1. dynamically, by the boot program, which sets the `siteNumber` field of the `bootConf` structure before invoking the kernel start entry
2. statically, by setting the `chorusSiteId` kernel tunable in the ChorusOS system image built on the host:

```
% configurator -set chorusSiteId=n
```

n is the site number assigned to the target board. This number can be specified in hexadecimal, by prefixing the number with `0x`, or decimal.

When the site number is set dynamically, it is the responsibility of the boot program to determine the site number of the target. The method by which the site number is found by the boot program is fully boot dependent, and specific to the target board. It may, for example, be stored in NVRAM, dynamically generated from a unique board identifier. When the target is booted with the standard ChorusOS network boot monitor, the whole IP address used by the boot monitor is provided as the site number of the target.

When the site number is set statically, the site number is fixed within the system image. This approach is less flexible than the dynamic method because the same system image cannot be booted on similar target boards. A system image with a unique site number must be built for each target. For this reason, it should only be used when there is no way for the boot program to determine the site number of the target board.

Note - The value of the site number set with the `chorusSiteId` tunable takes precedence over the value of the site number provided by the boot program.

The site number is set to zero by default. If the `IPC_REMOTE` feature has been enabled, and the site number remains at zero, the following message is displayed on the system console:

```
WARNING - LOCAL SITE ID. NOT SET => REMOTE IPC disabled
```

Only local IPC communications are enabled if the site number has not been set.

Specific IPC Configuration

Remote IPC over Ethernet Data-link

To configure the remote IPC over Ethernet feature, set the `IPC_REMOTE` feature to `true` and the `IPC_REMOTE_COMM` feature to `EXT` (mentioned in “IPC Feature Configuration” on page 235). In addition, switch on the `IOM_IPC` feature:

```
% configurator -set IOM_IPC=true
```

This adds the Ethernet-specific module into the IOM component which acts as the IPC Ethernet data-link driver.

Once you have built and booted the ChorusOS system image on the target board `tgtbd1`, the IPC Ethernet data-link can be dynamically started. Use the built-in `ethIpcStackAttach` command of `C_INIT` (running on the target board `tgtbd1`) :

```
% rsh tgtbd1 ethIpcStackAttach ethernet-device-name
```

The *ethernet-device-name* argument is the name of your Ethernet device with which your remote IPC stack will communicate. This name is the full pathname of the Ethernet device in the target device tree, displayed on the target system console by the system at boot time. For example, a `genesis2` board with a `dec21140` Ethernet

controller connected through a raven PCI bridge, has the pathname `/raven/pci1011,9@e,0`. This argument is only needed when the target board has more than one Ethernet controller.

See the `ethIpcStackAttach(1M)` man page for more details.

The following example describes how to build a ChorusOS system image for two similar PowerPC-based target boards, `tgtbd1` and `tgtbd2`, each with an Ethernet controller. Site numbers must be unique and statically configured in each ChorusOS system image.

1. Configure Remote IPC over Ethernet, if not already configured:

```
% configurator -set IPC=true
% configurator -set IPC_REMOTE=true
% configurator -set IPC_REMOTE_COMM=EXT
% configurator -set IOM_IPC=true
```

2. Assign a site number to `tgtbd1`, then build and uniquely identify the ChorusOS system image:

```
% configurator -set chorusSiteId=1
% make chorus
% mv chorus.RAM chorus.RAM.tgtbd1
```

Assign a site number to `tgtbd2`, then build and uniquely identify the ChorusOS system image:

```
% configurator -set chorusSiteId=2
% make chorus
% mv chorus.RAM chorus.RAM.tgtbd2
```

3. Once you have booted the `chorus.RAM.tgtbd1` system image on `tgtbd1` and the `chorus.RAM.tgtbd2` system image on `tgtbd2`, run the `ethIpcStackAttach` command:

```
% rsh tgtbd1 ethIpcStackAttach
% rsh tgtbd2 ethIpcStackAttach
```

Applications which need to communicate through remote IPC can now be launched on the `tgtbd1` and `tgtbd2` targets.

Remote IPC over VME Bus

In current implementation of IPC over VME bus the following constraints must be satisfied:

- The kernel must be configured for remote IPC feature over the VME bus (see “IPC Feature Configuration” on page 235).
- Each ChorusOS system image for every VME bus board must have the same logical and uniform view of all the devices present on the VME bus, through their respective device trees. As the device tree of each VME board will be different, a different ChorusOS archive should be built and booted on each target board. Typically, on `genesis2` targets, the file `src/bsp/powerpc/genesis2/src/boot/deviceTree.c` must be modified and recompiled for each different CPU board involved in the IPC protocol.
- IPC memory allocated to each CPU board must be in contiguous blocks of 64 Kilobytes on the VME bus.
- The ChorusOS system image must be booted first on the VME bus system controller, and then on other (non-system controller) boards, in any order.

The following example describes what the device sub-tree representing the VME bus on each board should be. It assumes that your target consists of three VME boards in cage slots 0,1, and 2, your VME bus system controller is located in slot 0, 64Kb of memory is used on each board for IPC, and VME memory dedicated to IPC is allocated as follows:

TABLE A-1 VME memory dedicated to IPC

	bridge I/O registers	IPC memory
board 0:	0x20000000-0x2000ffff	0x20030000-0x2003ffff
board 1:	0x20010000-0x2001ffff	0x20040000-0x2004ffff
board 2:	0x20020000-0x2002ffff	0x20050000-0x2005ffff

The device sub-tree representing the VME bus on each board is illustrated in Figure A-1.

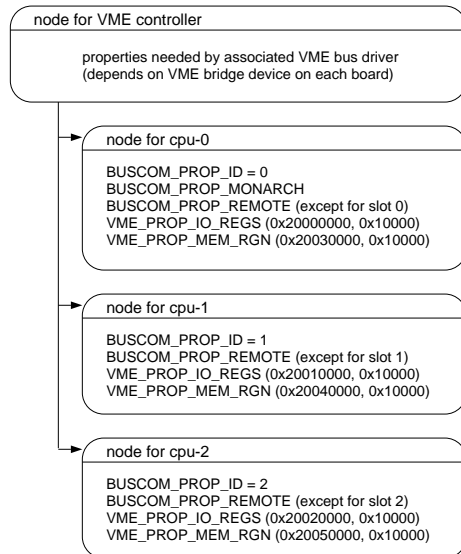


Figure A-1 Device sub-tree representing the VME bus

The main differences between ChorusOS system images for each board are:

- the properties associated with each VME bus controller device node
- the presence of the `BUSCOM_PROP_REMOTE` property in each child node. This property selects which local or remote instance of the bus communication (BUSCOM) VME device driver will be started on each node.

Building and booting ChorusOS system images on a VME target system is similar to the one described in the previous example. Embedding only the `genesis2` targets in a system image is achieved as follows:

- Edit the file `src/bsp/powerpc/genesis2/src/boot/deviceTree.c`.

Change `VME_MAX_BOARD` to 3:

```
#define VME_MAX_BOARD 3
```

- For each board:

- Change `LOCAL_CPU` to 0 for board 0, 2 for board 2, or 3 for board 3, for example:

```
#define LOCAL_CPU      0
```

- Compile the BSP, and build a ChorusOS system image using `make chorus`.
- Rename the ChorusOS system image to a file suffixed by the board number, for example `chorus.RAM.0`.
- Boot the system controller target first, using the system image suffixed with `.0`.
- Boot your other targets in any order, using the appropriate archive.

Glossary

actor	An actor is the unit of modularity for both applications and subsystems. The actor is also the unit of resource encapsulation used by the various ChorusOS operating system features. For example, at the Core Executive level, threads are resources attached to actors. Memory management features attach address spaces to actors.
actor capability	An actor capability is an unique handle, the possession of which grants the possessor the right to perform operations on the actor, such as modifying its address space, creating and deleting threads, removing it, and so forth. It is the concatenation of a Unique Identifier which is 64 bits long, and of a key which is also 64 bits long.
actor identifier	An actor identifier is a short 32-bit identifier used to identify dynamically loaded actors. Used as an argument to the <code>akill</code> and <code>await</code> commands. Only trusted actors may convert an actor identifier into an actor capability through the <code>acap()</code> service call.
aid	See actor identifier
application	An application is a program that enables you to do something useful with the ChorusOS operating system.
board	A board is another name for a simple target system.
boot actor	A boot actor is an actor which is loaded at boot time as part of the system image with the ChorusOS operating system.
boot server	A boot server is a system that stores the system image and makes it available for downloading by one or more target systems.

BSD	Berkeley Software Distribution is a version of the Unix operating system, developed at the University of California at Berkeley.
BSP	A Board Support Package is a set of target-specific files that contain information needed by the operating system to operate a particular board architecture.
core executive	The core executive is the central part of the ChorusOS operating system, with the ability to support multiple, independent applications running in both user and supervisor address space.
credentials structure	A credentials structure is a set of identifiers defining the privileges of a user.
debug server	A debug server is a program that is dedicated to providing information to a debugging tool in response to external requests.
device driver	A device driver is a software component that represents, to the operating system, a hardware component. The operating system interacts with the device driver to use the hardware component. This means that the operating system does not interact directly with the hardware of your system. For example, if your system communicates over Ethernet, your operating system must include an Ethernet device driver.
ELF files	Executable and Linking Format files are the dominant object, or executable, formats for UNIX.
ESDI	The Enhanced System Device Interface is an obsolete interface standard for hard disk drives.
event flags set	An event flags set is a set of bits in memory associated with a thread wait queue. Each bit, or event flag, is associated with an event. When the bit is set, the event is said to be posted, and the event is considered to have occurred. Otherwise the event has not yet occurred. A thread may wait on a conjunctive or disjunctive subset of the events in one event flags set.
FAT	A File Allocation Table is a hidden table in the MS-DOS file system of every cluster on a floppy or hard disk. The FAT records how files are stored in distinct, not necessarily contiguous, clusters (the basic unit of logical storage).
file system	In a computer, a file system is the way in which files are named and where they are placed logically for storage and retrieval. The

UNIX-based operating system has a file system in which files are placed somewhere in a hierarchical (tree) structure. A file is placed in a directory or subdirectory at the desired place in the tree structure.

FFS	The Fast File System was developed by the University of California at Berkeley to address performance problems with the existing file system.
flat memory	Flat memory is an implementation of memory management suited for platforms without an MMU. A single address space is managed, and virtual addresses match physical addresses. In this implementation, the memory management interface set is limited.
home actor	The home actor of a thread is the actor in which the thread was created, which may be different from the actor which created the thread.
host	A host is the system which provides services to the target, and where the development of the ChorusOS operating system and applications takes place.
IDE	Integrated Drive Electronics is a hard disk interface standard that offers high performance at low cost.
kernel	A kernel is the central module of an operating system. It is the part of the operating system that loads first, and remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications.
main thread	A main thread is the first thread running in an actor, created implicitly by the system in every boot actor, as well as in every dynamically loaded actor.
memory region	A memory region is a contiguous range of virtual addresses within an actor, treated as a unit by the memory system.
MMU	A Memory Management Unit is the part of a microprocessor responsible for mapping logical address space (as seen by programs) on to physical address space.
MS-DOS	Microsoft Disk Operating System is the standard, single-user operating system of IBM and IBM-compatible computers, introduced in 1981.

mutex	A mutex is a mutual exclusion lock. When this type of lock has been acquired and not released by a thread, another thread will be blocked and sleep if it tries to acquire the same lock. The thread will be awakened when the first thread releases the mutex.
NFS	The Network File System is a network file-access utility, developed by Sun Microsystems and subsequently released to the public as an open standard, that enables users of UNIX and Microsoft Windows NT workstations to access files and directories on other computers as if they were physically present on the user's workstation.
operating system	An operating system controls a computer system. It provides the common services used by every application running on the system, such as time keeping, or inter-process communication. In an embedded system, it is important to keep the operating system as small as possible while still providing all the services required. The ChorusOS operating system is configurable, so you do not need to include the components that are not necessary for your application or system.
owning actor	See home actor
PCI	The Peripheral Component Interconnect is a 32-bit expansion bus specification which supports Plug and Play.
PPP	The Point-to-Point Protocol is one of the two standards (the other is SLIP) for directly connecting computers to the Internet via dialup telephone connections. Unlike the older SLIP protocol, PPP incorporates superior data negotiation, compression, and error correction. PPP can also route non-IP traffic, for example IPX for Novell networks.
region descriptor	A region descriptor is a data structure used by actors to describe memory areas on which they want to perform a given operation. A region descriptor is only meaningful as part of a ChorusOS operating system invocation.
RTOS	A Real-Time Operating System is a system that responds to input immediately, rather than taking a few seconds, or minutes, to react. It offers response times which are predictable, or deterministic.
SCSI	The Small Computer Systems Interface is an interface in which you can plug devices such as hard disk drives, CD-ROM drives, scanners and laser printers.

secured mode	A secured mode is a mode of execution of the system, where checks are performed on credentials structures to determine the validity of an operation. In this mode, only the super user may load supervisor actors.
semaphore	A semaphore is an integer counter associated with a thread wait queue. Two atomic operations are available on semaphores: P (or <i>pass</i>) and V (or <i>free</i>). The P operation decrements the counter, and blocks the thread if the counter has reached a negative value. The V operation increments the counter and wakes up a thread, if any, in the semaphore wait queue.
SLIP	The Serial Line Internet Protocol is the earliest of two Internet standards (the other standard is PPP) specifying how a workstation or personal computer can link to the Internet by means of a serial line.
subsystem	A subsystem is a set of operating system components running above the micro-kernel which defines a self-contained, high-level API (or personality) for programs. The OS subsystem in ChorusOS, comprising the AM, IOM, ADMIN and C_INIT actors and associated libraries, defines the POSIX personality.
supervisor actor	A supervisor actor is a process, the threads of which are always supervisor threads. Supervisor actors only contain regions within the shared system address space.
super user	A super user is a privileged user whose identifier in the credentials structure is set to 0. When the system is running in secured mode, only the super user may load supervisor actors.
system actor	A system actor is a privileged process run by the ChorusOS operating system. A privileged user actor running in its own address space can also issue privileged requests.
system image	A system image is the binary or executable image that will be loaded on the target system. It includes the operating system and, optionally, application actors.
target	A target is the system where the ChorusOS operating system will run.
TCP	On the Internet, the Transmission Control Protocol is the protocol standard that permits two Internet-connected computers to establish a reliable connection.

thread	A thread is the flow of control within an actor. Each thread is associated with an actor and defines a unique execution state. An actor may contain multiple threads; the threads share the resources of that actor, such as memory regions or message spaces, and are scheduled independently.
thread identifier	A thread-identifier is a 32-bit context-dependent identifier, used to identify a thread within a given actor.
UDP	The User Datagram Protocol is one of the fundamental Internet protocols. UDP operates at the same level as the Transmission Control Protocol (TCP), but has a much lower overhead and is much less reliable. Unlike TCP, it does not attempt to establish a connection with the remote computer, but simply hands the data down to the connectionless IP protocol.
UFS	The UNIX File System, or UFS, designates the superior BSD-UNIX file system, as apposed to the System V file system.
user actor	A user actor is a process without any special privileges, run by the user. User actors have private user address spaces.
XIP	Execution In Place is the name given to object code that does not have to be relocated before being executed.
XRAY	The ChorusOS operating system debugger from Mentor Graphics Corporation.

Index

A

ACTOR_EXTENDED_MNGT

description of 35

actors

adding them to the system image using
 ews 81

boot

definition of 126

building 60

communicating between 173

context of 127

definition of 121

determining privilege of 125

dynamic linking example 115

dynamically loading 126

embedding 61

execution environment of 127

loading 126

multi-threaded 134

multithreaded 101

naming 123

spawning 130

supervisor 97, 123

terminating 130

types of 122

user 97, 123

address spaces

user and supervisor 123

ADMIN_CHORUSSTAT

description of 49

ADMIN_IFCONFIG

description of 49

ADMIN_MOUNT

description of 49

ADMIN_NETSTAT

description of 50

ADMIN_RARP

description of 50

ADMIN_ROUTE

description of 50

ADMIN_SHUTDOWN

description of 50

AF_LOCAL

description of 48

afexecve

example use of 131

allocating memory 161

allocating memory regions 163

API 100, 101

basic environment 98

Console Input/Output 66

extended environment 99

GNU 2.7.1 C++ 100

Mathematical 100

POSIX Input/Output 66

POSIX Micro Real-time Profile 100

Sun RPC 100

API, *see* application programming interfaces

application debugging

architecture 208

steps involved in 208

system debugging with 218

application programming interfaces

overview of 98

applications.xml 61

B

- basic configuration profile
 - feature settings 72
- basic environment
 - APIs in 98
 - description of 60
- basic profile
 - description of 51
- basic scheduling control 150
- Benchmarking
 - description of 54
- Bootmonitor
 - description of 54
- BPF
 - description of 46
- build environment
 - imake 204
 - mkmk 204

C

- C_INIT 63, 64
 - commands 67
 - description of 67
 - ethIpStackAttach 237
 - steps executed at system start-up by 68
- C_INIT authentication
 - reasons for failing 65
- C_INIT options
 - LOCAL_CONSOLE 44
 - RSH 44
- chadmin 211
- chconsole 212
- chls command 222
- chorusos.brd
 - description of 210, 218
- chorusos.brd file
 - description of 209, 218
- chserver 211
 - starting and configuring 212
- command-line configuration tool
 - see also* graphical configuration tool
- compilation options
 - make environment 103
- compiling and linking
 - information for 96
- configuration files 75
- configuration options

- types of 71
- configuration profiles
 - basic 72
 - extended 72
 - selecting 86
- configuration tools 76
- configurator 235
- core executive
 - description of 34
- creating threads 135

D

- DATE
 - description of 40
- dchls command, *see* chls command
- DEBUG_SYSTEM
 - description of 44
- DebugAgent
 - activating 206
 - tunable options in 206
- debugging
 - enabling for imake components 204
 - enabling for mkmk components 206
 - preparing for 203
 - symbolic 203
 - troubleshooting 222
- debugging architecture
 - debugging modes within 53
- debugging modes
 - application 53
 - system 53
- DebugServer 211
 - configuration file 213, 214
 - environment variable 213
 - slot numbers 212
 - starting 213
 - starting and configuring 212
 - stopping 214
- Default Console
 - description of 54
- default scheduler
 - CLASS_FIFO 36
- deleting threads 139
- descriptors
 - memory region 161
- /dev

- description of 68
- DEV_MEM
 - description of 46
- developing an application 55
- development environment
 - components of 51
- development lifecycle 54
- dlopen
 - example using 117
- dynamic applications
 - support for 113
- dynamic libraries
 - building 111
 - using 109
- dynamic parameters 74
- dynamic process management
 - see also* IPC
- dynamic programming 112
- dynamic programs
 - building 112
- DYNAMIC_LIB
 - see also* ACTOR_EXTENDED_MNGT

E

- environment
 - imake 104
 - make 103
- environment variables
 - adding using ews 80
 - changing using ews 80
 - deleting using ews 81
 - modifying using ews 80
 - runtime linker 114
- /etc/security 65
- EVENT
 - description of 38
- event flags 142
- ews
 - using 76
- executables
 - dynamic 110
 - relocatable 110
- extended configuration profile
 - feature settings 72
- extended environment
 - APIs in 99
 - description of 62

- running “Hello World” example in 65
- extended profile
 - description of 51

F

- feature
 - definition of 72
- feature options 72
- feature options, *see* configuration options
- feature settings
 - basic configuration profile 72
 - extended configuration profile 72
- features
 - ACTOR_EXTENDED_MNGT 35
 - ADMIN_CHORUSSTAT 49
 - ADMIN_IFCONFIG 49
 - ADMIN_MOUNT 49
 - ADMIN_NETSTAT 50
 - ADMIN_RARP 50
 - ADMIN_ROUTE 50
 - ADMIN_SHUTDOWN 50
 - AF_LOCAL 48
 - BPF 46
 - configuring using ews 80
 - DATE 40
 - DEBUG_SYSTEM 44
 - DEV_MEM 46
 - DYNAMIC_LIB 35
 - EVENT 38
 - FIFOFS 45
 - FLASH 47
 - FS_MAPPER 46
 - GZ_FILE 36
 - HOT_RESTART 37
 - IDE_DISK 46
 - IOM_IPC 47
 - IOM_OSI 48
 - IPC 41
 - IPC configuration 235
 - IPC_REMOTE 41
 - IPC_REMOTE_COMM 41
 - LAPBIND 42
 - LAPSAFE 43
 - LOCAL_CONSOLE 44
 - LOG 43
 - MIPC 41

- MON 43
- MSDOSFS 45
- NFS_CLIENT 45
- NFS_SERVER 45
- ON_DEMAND_PAGING 37
- PERF 43
- POSIX_MQ 42
- POSIX_SHM 42
- POSIX_SOCKETS 48
- PPP 48
- RAM_DISK 47
- ROUND_ROBIN 36, 151
- RSH 44
- RTC 40
- RTMUTEX 39
- SCSI_DISK 47
- SEM 38
- SLIP 48
- TIMER 39
- UFS 45
- USER_MODE 35
- using configurator to add 87
- using configurator to list 88
- using configurator to remove 87
- using configurator to view descriptions of 88
- VIRTUAL_ADDRESS_SPACE 37
- VTIMER 40
- VTTY 47
- FIFO scheduler 151
- FIFOFS
 - description of 45
- FLASH
 - description of 47
- freeing memory regions 163
- FS_MAPPER
 - description of 46
- function
 - dlopen 117

G

- glossary 241
- graphical configuration tool
 - see also* command-line configuration tool
- GZ_FILE
 - see also* ACTOR_EXTENDED_MNGT

H

- header files 101
- host file system, *see* root file system
- mounting
- hot restart
 - description of 29
- HOT_RESTART
 - description of 37

I

- IDE_DISK
 - description of 46
- /image/sys_bank
 - description of 68
- imake
 - build rules 104
 - building dynamic executables with 112
 - building dynamic libraries with 111
 - example using 107
 - packaging rules 106
 - using multiple source files with 108
 - variable definitions 104
- imake environment 104
- inter-process communication 40
- IOM_IPC 237, 238
 - description of 47
- IOM_OSI
 - description of 48
- IPC 40, 236, 238
 - configurations of 235
 - description of 31, 41, 187
- IPC_REMOTE 236 to 238
 - description of 41
- IPC_REMOTE_COMM 236, 238
 - description of 41

L

- LAP
 - description of 42
- LAP, *see* local access points
- LAPBIND
 - description of 42
- LAPSAFE
 - description of 43
- libraries 158

- choosing 97
- dynamic 109
- naming conventions of 98
- static 109
- linking
 - static and dynamic 110
- local access points
 - definition of 184
- Local Access Points, *see* LAP
- local IPC 235
 - configuring 236
- local IPC + remote IPC 235
 - configuring 236
- local IPC + remote IPC over VME 235
 - configuring 236
- LOCAL_CONSOLE
 - description of 44
- LOG
 - description of 43
- Logging
 - description of 54

M

- _main routine 97
- make environment 103
- management utilities 53
 - Benchmarking 54
 - Bootmonitor 54
 - Default Console 54
 - Logging 54
 - Monitoring 54
 - Profiling 54
 - Remote Shell 54
 - Resource Status 54
- MEM_FLAT
 - description of 36
- MEM_PROTECTED
 - description of 37
- MEM_VIRTUAL
 - description of 37
- memory
 - sharing 167
- memory management models
 - MEM_FLAT 36
 - MEM_PROTECTED 36
 - MEM_VIRTUAL 36
- memory protection

- description of 29
- memory region descriptors 161
- memory regions
 - allocating 163
 - freeing 163
- message pools
 - allocating messages from 177
 - definition of 174
- message queues 174
 - getting messages from 179
 - posting messages to 178
 - use of 179
- message spaces 174
 - creating 176
 - opening 176
- messages
 - definition of 174
- MIPC 41
 - description of 41
- MON
 - description of 43
- Monitoring
 - description of 54
- mounting
 - root file system 64
- mounting, *see* root file system
 - host file system
- MSDOSFS
 - description of 45
- multi-threaded actors 134
- mutexes 142, 147

N

- naming actors 123
- networking
 - see also* AF_LOCAL
 - see also* POSIX_SOCKETS
 - see also* PPP
 - see also* SLIP
- NFS_CLIENT
 - description of 45
- NFS_SERVER
 - description of 45
- non-secured mode
 - description of 65

O

ON_DEMAND_PAGING

description of 37

operating system

how constants are defined 96

how data types are defined 96

how error codes are defined 96

operating system components

description of 31

P

parameters

dynamic 74

static 74

tunable 74

PERF

description of 43

performance profiling

analyzing reports 229

clock 232

compiler options 227

description of 225, 231

feature options needed for 227

full form 226

generating reports 229

library 232

server 232

simple form 226

starting a session 228

stopping a session 228

tool set 231

policies

CLASS_FIFO 151

CLASS_RR 151

port numbers 209

POSIX_MQ

description of 42

POSIX_SHM

description of 42

POSIX_SOCKETS

description of 48

PPP

description of 48

profiles

basic 51

extended 51

Profiling

description of 54

program entry points 97

R

RAM_DISK

description of 47

RDBC 208

configuring and using 209

rdbc, *see* RDBC

RDBS 210

rdbbs 212

RDBS

configuring and using 217

rdbbs, *see* RDBS

remote IPC over Ethernet

configuring 237

remote IPC over VME

configuring 239

Remote Shell

description of 54

Resource Status

description of 54

root file system

mounting 64

ROUND_ROBIN

description of 36

RSH

description of 44

rsh command

available options of 63

communicating with the target using 63

RTC

description of 40

RTMUTEX

description of 39

runtime linker

environment variables 114

functions performed by 113

supported features 115

S

scheduler

description of 36

scheduling

threads 138

- scheduling policy
 - description of 36
- scheduling threads 150
- SCSI_DISK
 - description of 47
- secure mode
 - configuring the operating system in 64
- SEM
 - description of 38
- semaphore synchronization objects 38
- semaphores 142
 - atomic operations available on 38
 - description of 38, 143
- serial device
 - identifying 212
- sharing memory 167
- site number
 - description of 236
- SLIP
 - description of 48
- slot numbers 209, 218
- _start routine 97
- static parameters 74
- Sun Embedded Workshop
 - architecture diagram 28
 - components in 26
 - supported processor families 26
- supervisor actors 123
- synchronizing threads 142
- sysadm.ini
 - description of 63
 - example commands of 68
- system debugging
 - application debugging with 218
 - architecture 210
 - steps involved in 211
- system environment
 - using configurator to modify 90
- system image
 - adding actors to 81
 - configuring using ews 79
 - embedding actors into 61
 - rebuilding 62
 - rebuilding using ews 85
- system images
 - chorus 59
 - kernonly 59

T

- targets
 - deactivating 216
 - permanently removing 217
 - reactivating 217
 - registering 214
 - updating information 215
- thread scheduling 138
- thread semaphores 142
- threads 133, 158
 - creating 135
 - deleting 139
 - obtaining local identifier 135
 - per-thread data 154
 - scheduling 150
 - synchronizing 144
 - tools for synchronizing 142
- time management
 - options 39
- time management services
 - list of 191
- TIMER
 - description of 39
- timer services 193
- timers
 - arming 194
 - using 195
- Tools support 43
- tunable parameters 74
- tunables
 - configuring using ews 80
 - using configurator to change 89
 - using configurator to list 89
 - using configurator to view descriptions of 89

U

- UFS
 - description of 45
- user actors 123
- USER_MODE
 - description of 35

V

- VIRTUAL_ADDRESS_SPACE

description of 37
VTIMER
description of 40
VTTY
description of 47

W

waiting threads 143

X

XRAY 203, 208, 212, 218
example session with RDBS 219
start-up script 223