RadiSys ARTIC960 Programmer's Guide

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com

December 2000

References in this publication to RadiSys Corporation products, programs, or services do not imply that RadiSys intends to make these available in all countries in which RadiSys operates.

Any reference to a RadiSys licensed program or other RadiSys product in this publication is not intended to state or imply that only RadiSys Corporation's program or other product can be used. Any functionally equivalent product, program, or service that does not infringe on any of RadiSys Corporation's intellectual property rights or other legally protectible rights can be used instead of the RadiSys product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by RadiSys, are the user's responsibility.

RadiSys may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquires, in writing, to:

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(561) 981-3200

EPC, iRMX, INtime, Inside Advantage, and RadiSys are registered trademarks of RadiSys Corporation. Spirit, DAI, DAQ, ASM, Brahma, and SAIB are trademarks of RadiSys Corporation.

† All other trademarks, registered trademarks, service marks, and trade names are the property of their respective owners.

December 2000

Copyright © 2000 by RadiSys Corporation

All rights reserved

Contents

About This Book vii
- Contents Description vii
- Notational Conventions viii
- Terms viii
- Where To Get More Information ix
  - Reference Publications ix
  - Developer's Assistance Program x

Summary of Changes xi
- November 1998 xi
- March 1998 xi

Chapter 1: ARTIC960 Overview 1
- Supported Adapters 2
- Kernel 2
- On-Card STREAMS Environment 3

Chapter 2: Kernel Process Management 5
- Processes 5
- Process Scheduling 5
- Process States 6
- Process Initialization 6
- Process Termination 7
- Process Instance Data Services 7
- Spawning Processes 7
- Process Memory Protection 8
- Summary 9

Chapter 3: Kernel Device Drivers and Subsystems 11
- Device Driver/Subsystem Initialization 11
- Device Driver/Subsystem Access 12
  - OpenDev 12
  - InvokeDev 13
  - CloseDev 13
- Interrupt Handlers 13
- Vector Sharing 14
- Memory Protection 14
- Summary 15

Chapter 4: Kernel Resources 17
- Resource Management 17
  - Software Resources 17
  - Hardware Resources 18

Memory Management 18
Allocation 18
Suballocation 21
Dynamic Memory Allocation 22
Data Cache 22
Big-Endian Memory Addressing 23
Internal Data RAM 23
Summary 24
Process Synchronization 24
Semaphores 25
Events 26
Summary 27
Process Communication 27
Queues 28
Mailboxes 28
Signals 32
Summary 33
Timer Support 34
Software Timers 34
Time of Day 34
Performance Timer 35
Summary 35
Hooks 35

Chapter 5: Kernel Asynchronous Events 37
Asynchronous Events Notification 37
Terminal Error Notification 40
Exception Dependent Data Structures 41

Chapter 6: Kernel Trace Services 47
Trace APIs 47
printf C Function 47
Summary 48

Chapter 7: Kernel Commands 49
Using Kernel Commands 51
Summary 52

Chapter 8: System Unit Support 53
Implementing API Functions 53
Base API Services 54
Summary of Base API Services 55
Mailboxes 55
Utilities 56
Application Loader 56
Reset Utility 57
Configuration Utility 57
Status Utility 57
Dump Utility 57
Trace Utilities 58
Diagnostics Utility 58

Chapter 9: Compiling and Linking Programs 59
ARTIC960 Programs 59
OS/2 System Unit Programs 60
AIX System Unit Programs 60
Windows NT System Unit Programs 61
Index 63

About This Book

This book contains information about the ARTIC960 services available for writing adapter-resident programs. It also contains a brief description of the system unit utility programs and the steps required to compile and link both system unit and adapter programs. The book does not include sample code.

Contents Description

The following lists the contents of this guide.

Chapter		Description
1	ARTIC960 Overview	Describes the ARTIC960 adapter, including hardware, software, and supported adapters.
2	Kernel Process Management	Describes the kernel's view of processes and describes many of the process management services provided by the kernel.
3	Kernel Device Drivers and Subsystems	Describes the kernel device drivers and subsystems, including access calls, interrupt handling, and memory protection.
4	Kernel Resources	Describes the kernel resources, including hardware and software.
5	Kernel Asynchronous Events	Provides information on conditions than can occur on the adapter.
6	Kernel Trace Services	Describes the APIs and function to capture information about adapter activity.
7	Kernel Commands	Describes using kernel commands.
8	System Unit Support	Provides information on the base API services, mailboxes, and utilities.
9	Compiling and Linking Programs	Explains how to compile and link ARTIC960 programs and OS/2 system unit programs

Notational Conventions

This manual uses the following notations:

- `Screen text and syntax strings appear in this font.`
- All counts in this book are assumed to start at zero and all bit numbering conforms to the industry standard of the most significant bit having the highest bit number.
- All numeric parameters and command line options are assumed to be decimal values, unless otherwise noted.
- To pass a hexadecimal value for any numeric parameter, the parameter should be prefixed by **0x** or **0X**. Thus, the numeric parameters **16**, **0x10**, and **0X10** are all equivalent.
- All representations of bytes, words, and double words are in the little endian format.
- Utilities all accept the **?** switch as a request for help with command syntax.

Notes indicate important information about the product.

Cautions indicate situations that may result in damage to data or the hardware.

Tips indicate alternate techniques or procedures that you can use to save time or better understand the product.

ESD cautions indicate situations that may cause damage to hardware from electrostatic discharge.

The globe indicates a World Wide Web address.

Warnings indicate situations that may result in physical harm to you or the hardware.

Terms

- *ARTIC960* refers to the RadiSys ARTIC960 environment and can refer to programs that run on the following adapters, or the adapters themselves.
  - *ARTIC960 PCI* refers to functions supported only on ARTIC960 PCI adapters.
  - *ARTIC960Rx* refers to functions supported only on the ARTIC960Rx adapter.
  - *ARTIC960Hx* refers to functions supported only on the ARTIC960Hx adapter.
  - *ARTIC960RxD* refers to functions supported only on the ARTIC960RxD adapter
  - *ARTIC960 MCA* refers to functions supported only on the ARTIC960 Micro Channel adapter.
- S*ystem bus* can refer to either the Micro Channel or PCI bus.

Where To Get More Information

You can find out more about ARTIC960 from these sources:

- **World Wide Web**: RadiSys maintains an active site on the World Wide Web. The site contains current information about the company and locations of sales offices, new and existing products, contacts for sales, service, and technical support information. You can also send e-mail to RadiSys using the web site.

  When sending e-mail for technical support, please include information about both the hardware and software, plus a detailed description of the problem, including how to reproduce it.

  To access the RadiSys web site, enter this URL in your web browser:
  `http://www.radisys.com`

  Requests for sales, service, and technical support information receive prompt response.

- **Other**: If you purchased your RadiSys product from a third-party vendor, you can contact that vendor for service and support.

Reference Publications

You may need to use one or more of the following publications for reference:

- *ARTIC960 Programmer's Reference*
- *ARTIC960 STREAMS Environment Reference*
- Operating and Installation documentation provided with your computer system
- *Guide to Operations* books for one of the following co-processor adapters:

  ARTIC960 Micro Channel adapter

  ARTIC960 PCI adapter

  ARTIC960Hx adapter

  ARTIC960Rx adapter

  ARTIC960RxD adapter

  Each book contains a description of the co-processor adapter, instructions for physically installing the adapter, parts listings, and warranty information.

IBM Publications

- *IBM Operating System/2* (OS/2) Version 3.0
- *IBM Advanced Interactive Executive* (AIX) Version 4.1 and 4.2
- *IBM AIX Version 4.x Kernel Extensions and Device Support, Programming Concepts*, SC23-2207
- *IBM XL C Language Reference*, SC09-1260
- *IBM Personal System/2 Hardware Reference*, S85F-1678

Intel Publications:

- i960 RP Microprocessor User's Manual
- i960 Rx I/O Microprocessor Developer's Manual
- i960 Hx Microprocessor User's Manual
- i960 Cx Microprocessor User's Manual

STREAMS Information:

For information about writing a STREAMS module or driver, refer to the IBM Web site:

`http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/progcomc/ch10_streams.htm`

AIX supports a subset of SVR4.2 STREAMS calls, and the on-card STREAMS subsystem supports a subset of AIX STREAMS.

Developer's Assistance Program

Programming and hardware development assistance is provided by the RadiSys Developer's Assistance Program (DAP). The DAP provides, via phone and electronic communications, on-going technical support—such as sample programs, debug assistance, and access to the latest microcode upgrades.

You can get more information or activate your *free* DAP membership by contacting us.

- By telephone, call (561) 981-3200
- By e-mail, send to artic@radisys.com

Summary of Changes

This section lists the changes made to this book for recent releases.

November 1998

For this edition, the changes to the *ARTIC960 Programmer's Guide* are as follows:

- AIX support for the ARTIC960RxD adapter had been added.
- ARTIC960 support for AIX can support a total of 14 adapters—card 0 through card 13.

Note: System unit mailboxes support remote communication only with adapters 0 through 9. Local mailboxes can be used on all adapters.

March 1998

For this edition, the changes included the ARTIC960 Support for Windows NT, Version 1.0.

- *Implementing API Functions* on page 53
- *Base API Services* on page 54
- *Mailboxes* on page 55
- *Windows NT System Unit Programs* on page 61

1 ARTIC960 Overview

The ARTIC960 adapters are 80960-based co-processor adapters that act as a hardware/software platform for the development of user applications. They are customized to meet specific applications by the addition of daughter boards, such as the PCI Mezzanine Card (PMC) or Application Interface Boards (AIBs), and user-developed software. Some typical applications consist of an AIB or PMC with communications (such as ISDN, SS7, frame relay, or X.25) and adapter-resident processes that perform hardware management, protocol conversion, data formatting, and data transmission to the system unit and/or other ARTIC960 adapters.

ARTIC960 hardware capability includes:

- Up to 32-MB memory
- Memory protection
- Hardware timers
- Two system channels
- An AIB or PMC (daughter cards)

The software provided for the ARTIC960 adapters is segmented into two parts: the part that runs on the adapter itself, called the *kernel*, and the part that runs on the system unit, called *system unit support*.

The kernel is a collection of executable files providing the following capabilities:

- Real-time multitasking kernel
- System unit-to-adapter process communications
- Adapter-to-adapter process communications

The system unit support is a collection of IBM Operating System/2 (OS/2) 3.0 (and higher), IBM Advanced Interactive Executive (AIX) Versions 4.1 or 4.2, and Windows NT Version 4.0 executable files that provide:

- Adapter status and configuration information
- System unit-to-adapter process communications
- Application loader
- Adapter dump facility
- Debug facilities

The ARTIC960 RxD is supported only on AIX.

Supported Adapters

Table 1-1 shows which adapters are supported by each operating system.

Table 1-1. Operating System Support for Adapters

Adapter	OS/2 Version 1.2.2	AIX Version 1.4.1	Windows NT Version 1.2.0
ARTIC960 Micro Channel	√	√	
ARTIC960 PCI	√	√	√
ARTIC960 Rx PCI	√	√	√
ARTIC960 Hx PCI	√	√	√
ARTIC960 RxD PCI		√	
ARTIC960 Rx Frame Relay PCI	√		√

Kernel

The ARTIC960 kernel is a real-time multitasking kernel designed for high performance, but also with usability and portability as important objectives. It has a high level of function with extensive process management, process synchronization, and process communication support, as well as device driver, timer support, and asynchronous error notification capabilities. The kernel consists of the following executables:

Executable	Description
ric_kern.rel	Base kernel
ric_kdev.rel	Base kernel (development version)
ric_base.rel	Memory protection portion of the base kernel
ric_mcio.rel	Input/output subsystem
ric_scb.rel	Peer-to-peer transport subsystem
ric_oss.rel	STREAMS subsystem
ric_ess.rel	STREAMS cross-bus driver
ric_pci.rel	PCI local bus configuration device driver

Each of these files is a relocatable image, which is downloaded to the adapter's random access memory (RAM). Some of the files can be passed initialization information when they are loaded. For example, ric_kern.rel accepts a set of kernel configuration parameters that define how it will operate. Chapters 2 through 7 describe the capabilities provided by the kernel.

Refer to the *ARTIC960 Programmer's Reference* for information on the base kernel development version and the PCI local bus configuration device driver.

On-Card STREAMS Environment

The ARTIC960 runtime environment provides the standard UNIX System V Release 3 and 4 STREAMS tool set for running STREAMS-based modules and drivers on the ARTIC960 co-processor. For reference, use the following publications:

- Programming:

  STREAMS Modules and Drivers
  Unix System V Release 4.2
  ISBN 0-13-066879

- Reference:

  Device Driver Reference
  Unix System V Release 4.2
  ISBN 0-13-042631-8

Contact Prentice Hall at (515) 284-6761 to order single copies of the documentation.

Contact the Corporate Sales Department at (201) 592-2863 to make a bulk purchase in excess of 30 copies.

Benefits associated with STREAMS on the ARTIC960 co-processor include the following:

- Offloads the system unit from running communication protocol stacks by downloading protocol stacks to the ARTIC960 co-processor.
- Allows STREAMS-based protocol drivers written under UNIX STREAMS System V Release 3 and 4 specification from a UNIX or non-UNIX operating system to run under the ARTIC960 Kernel environment.
- Provides a flexible, portable, and reusable set of tools for development of system communication services following a widely distributed standard in the industry.
- Allows easy creation of independent modules that offer standard data communications services and the ability to manipulate those modules on a stream.
- From the system unit, on-card drivers can be dynamically loaded and interconnected (linked) on the ARTIC960 adapter making it possible to connect protocol stack drivers from various vendor sources.

To provide on-card STREAMS access and services to system unit applications, the On-Card STREAMS Environment consists of the following major parts:

- A system unit component called *STREAMS Access Library* (SAL) provides the access to the On-Card STREAMS Environment through an application program interface (API) that provides easy and unrestricted access to the On-Card STREAMS Environment from both UNIX- and non-UNIX-based operating systems.
- An on-card component called *On-Card STREAMS Subsystem (OSS)* provides the UNIX System V Release 3 and 4 STREAMS tool set in the ARTIC960 adapter.
- An on-card component called *On-Card STREAMS Cross Bus Driver (ESS)* provides support to transmit STREAMS data across the system bus between SAL and OSS.

For more information on OSS, refer to the *ARTIC960 STREAMS Environment Reference*.

2 Kernel Process Management

This chapter describes the kernel's view of processes and describes many of the process management services provided by the kernel. A complete list is on page 9, and a detailed description of each is in the *ARTIC960 Programmer's Reference*.

Processes

An ARTIC960 process is one or more programs bound together into a single executable that can be run on the ARTIC960 adapters. It can be downloaded onto the adapters by the application loader (RICLOAD) or spawned (created) by an existing process. Processes consist of the following components/attributes:

- Code, data, stack, and optional load parameters
- Process name, process ID, priority
- Resources allocated, such as memory, semaphores, and mailboxes
- Optional signal handler, exit handler, or asynchronous notification handler

There is no fixed limit on the number of processes that the kernel supports. The maximum number is a function of the memory available and the total number of resources allocated. A process is uniquely identified by its name (supplied when it is loaded or created) or its process identification (ID). Most kernel application program interface (API) calls use the process ID. The kernel service QueryProcessStatus resolves the process name to the process ID as well as providing other status information.

Process Scheduling

Process scheduling is priority based and preemptive. Each process has a priority, and the kernel ensures that the highest-priority process (that is ready to run) runs. In addition, when a process becomes ready to run that has a higher priority than the currently executing process, the current process is preempted and the new process is dispatched. QueryProcessInExec returns the process ID of the currently executing process.

Priority levels are from 0 to 255, with 0 the highest and 255 the lowest priority. Levels 0 through 15 are reserved for the kernel and its subsystems; levels 16 through 31 are reserved for user subsystems. Priority level can be set at load time, through a kernel service, or by default with a value set in the kernel configuration file. The kernel services QueryPriority and SetPriority calls dynamically return and set priority levels, respectively.

A time-slice timer is used to guarantee sharing of the CPU among processes running on the same priority level. In this case, the processes share the CPU in a round-robin fashion. The time-slice timer has a configurable granularity (through the kernel configuration file) that defaults to 10 milliseconds and can be totally disabled.

Because processes can be preempted at any time, the kernel services EnterCritSec and ExitCritSec are provided to disable and enable preemption around critical sections of code. These services also allow interrupts to be enabled and disabled. They maintain a depth count so that critical sections can be easily nested. For example, a routine called from within another routine's critical section can perform an EnterCritSec and ExitCritSec pair, and preemption (or interrupts) remains disabled until the calling routine performs its ExitCritSec. A process cannot, however, keep preemption and/or interrupts disabled across dispatch cycles. If a process blocks itself or calls a service that blocks, the kernel enables interrupts or preemption, or both, and resets the depth count.

Process States

At any given time, a process is in one of the following states:

- Stopped
- Ready
- Blocked
- Running
- Suspended
- Stopping

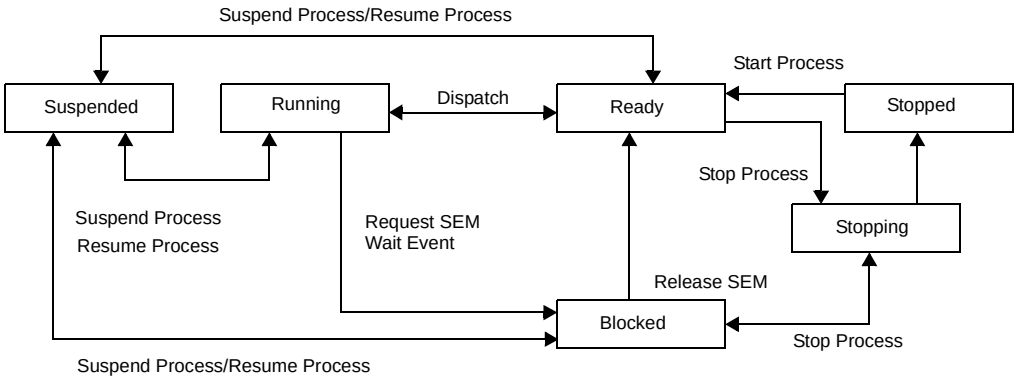


Figure 2-1. Process State Diagram

Process Initialization

When a process begins execution, it performs any required initialization, such as memory and other resource allocation. Once initialization is complete, it may issue a CompleteInit call to the kernel. This call allows the process to report its initialization status to the kernel. This status can be retrieved by the application loader and the status utility (for more information on these utilities, see *Utilities* on page 56). If a process reports an error code value of zero, CompleteInit returns to the calling process. If a non-zero error code is provided, the process is stopped, and any resources it acquired are freed.

Process Termination

A process can be stopped or unloaded. When it is stopped, it remains intact on the adapter, but any resources it acquired are freed by the kernel. When a process is unloaded, it is removed from the adapter and its resources are freed.

A process can be stopped by:

- Calling StopProcess or UnloadProcess—either from itself or by another process.
- The C startup library routine. If a process is coded such that it returns from main(), either through *return*, by a closing brace, or by exit().
- Calling the C exit library routine.
- Calling CompleteInit with a non-zero error code.
- The kernel, if the process causes certain fault conditions. For more information, see *Chapter 5: Kernel Asynchronous Events* on page 37.

A stopped process can be restarted using the StartProcess call, the RICLOAD utility, or the kernel Command facility (see *Chapter 7: Kernel Commands* on page 49).

A process is unloaded only when it, or another process, calls UnloadProcess.

A process can register a routine to be called when it terminates through the SetExitRoutine service.

An exit handler may call most, but not all, kernel services. (See the *ARTIC960 Programmer's Reference* for the restrictions.)

Process Instance Data Services

The SetProcessData and GetProcessData services provide a means for an application driver to define process-dependent data and to retrieve the data when needed.

The process data services maintains pointers to process instance data for up to 15 application IDs per process. Application IDs 0—63 are reserved for ARTIC960 kernel.

An application that needs to make use of process instance data should define a single structure to contain all the data needed by all services in that application environment.

Spawning Processes

A process can be spawned through CreateProcess. It creates a new process that shares the code and data areas of the caller. The new process is actually a peer of the creating process; it is *not* a child process. If the creating process dies, the new process is unaffected. Additionally, the new process does not have access to any of the resources that the creating process acquired. If access is needed, the new process must *open* the resources itself. Even if the creating process is a device driver or subsystem, the new process will be a normal process. (See *Chapter 3: Kernel Device Drivers and Subsystems* on page 11 for more information.)

Process Memory Protection

The development version of the kernel supports process memory protection on the ARTIC960 and ARTIC960 PCI adapters. It is optional and is controlled through the `MEMORY_PROTECTION` parameter in the kernel configuration file. When turned on, all processes run with memory protection (with the possible exception of subsystem/device drivers). It is expected that memory protection will be used primarily as a debug facility.

When running with memory protection, processes cannot address memory or memory mapped I/O locations that they have not acquired. If they want to share memory with another process, they must access that memory using the OpenMem kernel call. If a process attempts to address memory it does not own, it is stopped.

Memory protection for processes is accomplished by maintaining a memory protection map for each executing process. The map contains access right information for each page of memory in the system. When memory protection is enabled, hardware checks each address issued by the processor against the protection map. If the process does not have the proper access, the kernel is notified through a high-priority interrupt (trap).

Summary

The Process Management Services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
CompleteInit	Notifies the kernel that the calling process has completed initialization. This call can also indicate initialization errors.
QueryProcessStatus	Gets the status and other related information for a process. This call can also be used to resolve a process name into a process ID.
SetExitRoutine	Sets the exit routine for the process. The kernel calls an exit routine of the process, if the process is to be stopped or unloaded. The exit routine can be used for any cleanup that is required.
QueryCardInfo	Gives level information of software and hardware.
QueryConfigParams	Returns the kernel configuration parameters.
CreateProcess	Creates a new process. A set of parameters can be passed to the new process. The new process shares the creator's code and data, but gets its own stack. The new process does not get access to any other resources acquired by the creator.
StartProcess	Starts a stopped process.
StopProcess	Stops a started process. All resources acquired by the process are released. The memory for the process code, data, and stack is not released so the process can be restarted later.
UnloadProcess	Unloads the process. All the resources acquired by the process are released. The memory for the process code, data, and stack is also released.
SuspendProcess	Takes a process off the dispatch queue and suspends it.
ResumeProcess	Queues the process on the dispatch queue.
QueryProcessInExec	Returns the process ID of the process currently executing.
SetPriority	Changes the priority of the current process.
QueryPriority	Queries the priority of a process.
SetProcessData	Defines process instance data.
GetProcessData	Retrieves process instance data.
EnterCritSec	Disables interrupts or preemption selectively for the caller.
ExitCritSec	Enables interrupts or preemption selectively for the caller.
Dispatch	Causes the next process on the same priority level to run, if ready.

3 Kernel Device Drivers and Subsystems

Device drivers and subsystems are a special class of processes that act as service providers. They are used to insulate the application writer from hardware details or to implement new classes of services. Unlike normal processes, they have access to privileged kernel services to allocate hardware resources, interrupt vectors, and to modify memory protection rights. Page *15* lists all services related to device drivers and subsystems. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each.

Device drivers do not have a process-time component. They run only during initialization, through an interrupt or a request for service by another process. Subsystems have the capabilities of device drivers but, additionally, have their own process time.

Processes access device drivers/subsystems by performing an OpenDev call. Access is removed by the CloseDev call. Actual calls to a device driver or subsystem are made using the InvokeDev call.

Device Driver/Subsystem Initialization

When processes are first loaded onto the adapter, they have normal process status. To become a subsystem or device driver, a process must declare itself to be a device driver or a subsystem by calling the CreateDev service. This service specifies to the kernel whether the caller is to be a device driver or subsystem, as well as specifying the open, close, and call entry points, and whether the process should run with memory protection on or off. At this point, the process has established itself as a device driver or subsystem and can use the following privileged kernel services, in addition to the normal services:

- SetProcMemProt
- QueryProcMemProt
- AllocVector
- ReturnVector
- SetVector
- AllocHW
- ReturnHW
- QueryHW
- AllocVectorMux
- SetVectorMux

During initialization, resources such as memory, queues, semaphores, and hardware (which are to be owned by the device driver/subsystem) should be allocated. (For more information on resources, see *Chapter 4: Kernel Resources* on page 17.)

When initialization is complete, CompleteInit is called—which tells the kernel that the device driver or subsystem can now be called by other processes. Because a device driver has no process time of its own, it does not regain control after the call to CompleteInit.

Device Driver/Subsystem Access

Any process (including device drivers and subsystems) can access a device driver or subsystem by way of the OpenDev, InvokeDev, and CloseDev calls. Each of these calls results in the specified subsystem/device driver getting control. There are no restrictions on the use of kernel services while executing in a subsystem or device driver. However, because it is running on the caller's process time, any resource allocated belongs to the calling process. If the device driver needs to own a resource, it must be allocated at initialization. A subsystem can allocate resources it owns either at initialization or when it is running on its own process time.

OpenDev

To establish a connection with a subsystem/device driver, the requester issues an OpenDev call with the name of the subsystem or device driver (optionally, other parameters can be passed). The kernel resolves this name to a subsystem/device driver open entry point and calls that entry point. The subsystem/device driver can then perform any instance initialization required. Any resources allocated at this time belongs to the requesting process. If the device driver/subsystem accesses resources on behalf of the calling process, it must open the resource with the requesting process's context.

When the subsystem/device driver returns to the kernel, the kernel provides a handle to the requester, which is passed on subsequent InvokeDev and CloseDev calls.

A correlation value called DevMemo can be passed back to the kernel by the subsystem/device driver when returning from the open call. This value is provided by the kernel back to the subsystem/device driver when the opener issues an InvokeDev. The content of this variable is implementation defined, but is primarily intended as a pointer or an index to aid the subsystem/device driver in locating instance-specific information. For example, a device driver may need to initialize a control block with some opening process-specific information. By returning a pointer to this control block in the DevMemo field, the device driver can easily access it on subsequent InvokeDev calls.

Refer to the *ARTIC960 Programmer's Reference* for additional information on CreateDev and OpenDev.

InvokeDev

Processes actually request service through the InvokeDev call. This call accepts a handle and a pointer to a subsystem/device driver defined parameter control block.

When InvokeDev is issued, the subsystem/device driver gets control at its call entry point. A pointer to the parameter block, parameter block size, process ID, and devmemo information is provided by the kernel.

Refer to the *ARTIC960 Programmer's Reference* for additional information on CreateDev and InvokeDev.

CloseDev

The CloseDev call removes the connection to a subsystem/device driver previously established by OpenDev. The subsystem/device driver should free any resources previously allocated when it was opened by the requester.

If CloseDev is issued by a subsystem or device driver, the kernel removes the connection between it and all processes that had previously opened it. Further attempts to access the subsystem/device driver will fail.

Refer to the *ARTIC960 Programmer's Reference* for additional information on CreateDev and OpenDev.

Interrupt Handlers

Device drivers and subsystems may require interrupt handlers to perform their function. The kernel is made aware of the need for an interrupt handler through the AllocVector service.

The requester specifies the interrupt vector number it wants, the interrupt handler entry point, and whether the interrupt handler should be called with or without memory protection.

When an interrupt occurs, the kernel's first-level interrupt handler saves the environment of the interrupted process (or interrupt handler) and then calls the subsystem/device drivers interrupt entry point. At this time, interrupts are enabled and the processor is executing at the priority level of the interrupt. Higher-priority interrupts can be received but none at the same- or lower-priority level. Interrupts can be totally enabled or disabled using the enter (EnterCritSec) and exit (ExitCritSec) critical service calls described on page 9.

Most kernel services are available to interrupt handlers. However, there are some restrictions. For instance, resources cannot be allocated or freed. (See *Chapter 4: Kernel Resources* on page 17, for information on resources.) Nor can calls be made that require process blocking, such as SuspendProcess. Refer to the *ARTIC960 Programmer's Reference* for a list of services and their interrupt time-usage restrictions.

Vector Sharing

ARTIC960 Support for AIX, Version 1.2, and ARTIC960 Support for Windows 1.0 support sharing of interrupt vectors on the adapter. Two new kernel services have been added to support this feature. AllocVectorMux and SetVectorMux allow allocating/resetting a handler for a shared vector. The handlers must return a value to indicate whether the interrupt was claimed. A macro, SetInterruptPriority, has also been added. It allows an interrupt handler to lower its priority to allow other interrupts at the same level to be serviced.

All vectors that are registered for a shared interrupt are called when the interrupt occurs. The order in which vectors are called is unspecified.

Memory Protection

Three memory protection options relate to subsystems/device drivers:

- Global memory
- Subsystem/device driver process-time and call memory protection
- Subsystem/device driver interrupt-time memory protection

The global memory option is controlled by the parameter `MEMORY_PROTECTION (YES|NO)` contained in the kernel parameter file. The second and third options are contained in the CreateDev and AllocVector calls, respectively.

These options control whether memory protection is in effect when the subsystem/device driver is called, either as an extension of the calling process, by the dispatcher on its own process time, or at interrupt time. The options are hierarchical in nature. The global option must be on for the process/call option to have any effect, and the process/call option must be on for the interrupt time option to have any effect.

Memory-Protection Maps

When a subsystem/device driver is called with memory protection enabled, the kernel switches to that subsystem/device drivers' memory-protection map and adds access to the parameter block passed on the call—allowing access to the parameter block by the subsystem/device drivers. When the called subsystem/device driver returns to the kernel, the parameter block is unmapped, and the kernel switches back to the calling process's memory protection map.

Any addresses passed in the parameter block are not mapped by the kernel. The memory associated with these addresses have to be explicitly mapped and unmapped using the SetProcMemProt call. Device drivers and subsystems should check memory access rights on these addresses if the caller is running with memory protection. In particular:

- Input data pointers should be checked for read access by the caller
- Output data pointers should be checked for read/write access by the caller
- Code pointers should be checked for read access.

In addition, device drivers and subsystems that are running with memory protection need to get read/write access to return output parameters using user-provided pointers. This access needs to be dropped before returning to the caller. The access can be added and dropped using the SetProcMemProt service.

Dynamic Memory

If a driver or subsystem allocates dynamic memory in a handler, and wants to pass that memory address to a process that is running with memory protection active, it must explicitly give the process access to the allocated memory using the SetProcMemProt call.

Summary

The Device Driver/Subsystem Services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
CreateDev	Registers the calling process as a subsystem or device driver. This service also takes entry point addresses for subsystem and device driver call.
OpenDev	Opens a previously registered subsystem or device driver. The subsystem or device driver gets control at its open entry point.
CloseDev	Releases access to a registered subsystem or device driver. The subsystem or device driver gets control at its close entry point.
InvokeDev	Calls a subsystem or device driver at its strategy entry point. The subsystem or device driver gets control at its call entry point.
AllocVector	Allocates a set of interrupt vectors to the calling subsystem or device driver.
ReturnVector	Returns a set of interrupt vectors to the calling subsystem or device driver.
SetVector	Sets a new entry point for an allocated interrupt vector.
AllocHW	Allocates a hardware device to the calling subsystem or device driver.
ReturnHW	Returns a hardware device to the calling subsystem or device driver.
QueryHW	Returns the allocation status of a hardware device to the calling subsystem or device driver.
AllocVectorMux	Allocates a shared interrupt vector to the calling subsystem or device driver.
SetVectorMux	Sets a new entry point for allocated shared interrupt vector.

4 Kernel Resources

The kernel (which is downloaded to the adapter) supports both hardware and software resources. Software resources consist of the following:

- Memory
- Semaphores
- Events
- Queues
- Mailboxes
- Signals
- Timers
- Hooks

Hardware resources include:

- Hardware devices such as DMA channels and communication ports
- Vectors

Refer to the *ARTIC960 Programmer's Reference* for a detailed description of these resources.

Resource Management

The kernel provides services for managing resources.

Software Resources

A resource is acquired by issuing either a create or an open call for the resource that is to be accessed. The *create* can be used at any time to allocate a resource, whereas *open* is used when another process wants to share a resource previously created by another process. All software resources, except software timers, can be shared.

A resource must be identified by name if it is to be shared. If it is not to be shared, it can remain nameless. The name is a null-terminated American National Standard Code for Information Interchange (ASCII) string of up to 16 characters in length.

The create and open calls usually return a resource handle that is used for other services pertaining to the resource. A resource handle is unique to each requester of a resource. A process must create or open each resource that it accesses. The resource handle cannot be shared among peer processes.

To release access to a resource, processes use a close service.

In most cases, all processes that share a resource are peers, regardless of whether they used a create or open service to acquire the resource. When the last process closes a shared resource, the resource ceases to exist. Peer processes that share a resource can be notified through Asynchronous Event Notification when one of the peers is stopped or unloaded. See *Chapter 5: Kernel Asynchronous Events* on page 37 for more information.

Hardware Resources

Hardware resources are acquired by performing an allocate call for the particular resource and freed using a return call. Unlike software resources, hardware resources can never be shared.

Memory Management

The kernel supports three levels of memory management:

- **Allocation**
  The method a process uses to dynamically allocate large pieces of memory. It is always performed in page size (4K) increments.
- **Suballocation**
  Smaller allocation sizes can be handled using suballocation because it is an efficient management scheme for acquiring and releasing buffers from a block of memory already allocated by a process.
- **Dynamic memory allocation**
  Allows allocating and freeing of small blocks of memory from a dynamic memory pool.

A complete list of memory management services is shown on page *24*. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each.

Allocation

Memory can be dynamically allocated and deallocated using the CreateMem and CloseMem services. A process wanting to share memory with another process does so using the OpenMem call. When allocating memory, the requester specifies, among other things, the type of memory and the access rights of that memory.

Memory Type

ARTIC960 adapters can contain two separate banks of memory—instruction memory and packet memory. Instruction memory access is optimized for program code and local data. Packet Memory access is optimized for DMA data buffers. This approach maximizes performance by having less memory contention between the central processor unit (CPU) and direct memory access (DMA) channels. Memory access contention only occurs when the CPU and a DMA channel simultaneously attempt to access the same bank of memory.

Access Rights

Access rights define whether a memory block can be read or written, or both, and what source has the ability to access it. Potential sources are CPU, daughter card, and system bus, and they are defined as follows:

CPU Memory references resulting from code running in the processor—includes instruction fetches and memory reference instruction addresses.

Daughter Card DMA
Memory references either going to or coming from a daughter card.

System bus
Memory references to and from this adapter as a system bus slave, as well as memory references resulting from the adapter's system bus DMA channels.

A set of constants has been defined, which the caller of CreateMem and OpenMem can use to specify access rights. The access rights are listed on page *20*. They can be ORed together to achieve whatever access is wanted.

The access rights work slightly differently, depending on whether the access is a CPU access or a daughter card/system bus access. For CPU access, each process can specify the access rights it wants. For example, one process could create a block of memory named MEM1 and give it CPU read and write access. A second process could open MEM1 and give it only read access.

For daughter card/system bus, the access rights specified for a block of memory is based on all processes that requested access to that memory. For example, one process could create a block of memory named DMABUFFERS with system bus read access. A second process opens DMABUFFERS with system bus read and system bus write access. The `MEM_OVERRIDE_MC_ACCESS` (or `MEM_OVERRIDE_AIB_ACCESS`) must be specified when a second process opens a memory area, if the access to be changed is different from when the memory area was created). The memory access for DMABUFFERS would then be system bus read and write. If an opener of DMABUFFERS wants to set the access rights unconditionally and independently of other processes, it must specify `MEM_OVERRIDE_MC_ACCESS` along with the access rights it wants.

For the ARTIC960, hardware memory protection only validates CPU and system bus slave accesses. Daughter card DMA and system bus DMA addresses are validated by software prior to loading the DMA channels.

Memory Access-Right Constants

Constant	Description
MEM_SHARE	Memory is sharable with other processes. The default is that memory is not sharable.
MEM_READABLE	The memory can be read by the 80960. The default is the memory cannot be read or written by the 80960.
MEM_WRITABLE	The memory can be written by the 80960. The default is the memory cannot be read or written by the 80960.
MEM_OVERRIDE_MC_ACCESS	The current system bus access to the created memory is overridden. The default system bus (or daughter card) access is not changed.
MEM_MC_READABLE	Memory can be read from the system bus. In addition, the on-card system bus DMA can read the memory. The default is memory cannot be read or written from the system bus.
MEM_MC_WRITABLE	Memory can be written from the system bus. In addition, the on-card system bus DMA can write to memory. The default is memory cannot be read or written from the system bus.
MEM_OVERRIDE_AIB_ACCESS	The current daughter card access to the created memory is overridden. The default system bus (or daughter card) access is not changed.
MEM_AIB_READABLE	The daughter card DMA can read from the memory. The default is memory cannot be read or written by the daughter card DMA.
MEM_AIB_WRITABLE	The daughter card DMA can write to the memory. The default is memory cannot be read or written by the daughter card DMA.
MEM_DCACHE	Memory can be cached. The default is that memory cannot be cached.
MEM_BIG_ENDIAN	Memory is treated as big endian. By default, all memory is treated as little endian. Big-endian memory regions are supported only on the ARTIC960Hx adapter.

Memory Sharing

For two or more processes to share memory, one process must first create the memory. Subsequent sharing processes then open that memory. The memory must be named and created as sharable. The access rights on CreateMem and the OpenMem can be different. Processes should never attempt to bypass this sharing mechanism. It precludes using memory protection.

Using Memory Allocation

The following describes how Process A goes about sharing memory with Process B. Process A has CPU read/write, whereas Process B has CPU read access. The memory block also has daughter card read-only access.

- Process A does a CreateMem call with the following access rights:

  ```
  MEM_SHARE | MEM_READABLE | MEM_WRITEABLE | MEM_AIB_READABLE
  ```

- Process B does an OpenMem call with the access right of

  ```
  MEM_READABLE
  ```

If Process B wanted to modify the daughter card access to make it read/write instead of read only:

- Process A does a CreateMem call with the following access rights:

  ```
  MEM_SHARE | MEM_READABLE | MEM_WRITEABLE | MEM_AIB_READABLE
  ```

- Process B does an OpenMem call with the access rights of

  ```
  MEM_READABLE | MEM_OVERRIDE_AIB_ACCESS | MEM_AIB_READABLE |
  MEM_AIB_WRITEABLE
  ```

Suballocation

A block of memory can be suballocated into some number of smaller fixed-size blocks, using the InitSubAlloc call. One or more of these smaller blocks can then be allocated and freed, using GetSubAlloc and FreeSubAlloc. The GetSubAllocSize service is provided to calculate the required size of the larger block, given the number and size of the smaller blocks.

Because a portion of the larger memory block is used for management of the smaller blocks, the GetSubAllocSize returns the total size required, given the number and size of smaller blocks needed. Processes should use GetSubAllocSize to insulate themselves from the internal kernel suballocation management overhead. Otherwise, application breakage could occur on a later release of the kernel.

Using Memory Suballocation

The following describes how a process sets a block of memory for suballocation into a pool of 100 256-byte blocks:

- Issues GetSubAllocSize with a unit count of 100 and a unit size of 256.
- Does a CreateMem with a size equal to that returned on GetSubAllocSize.
- Issues InitSubAlloc with:
  - A pointer to the memory returned on CreateMem
  - Unit count of 100
  - Unit size of 256

The process can now perform GetSubAlloc and FreeSubAlloc calls to obtain buffers of 256 bytes or multiples of 256 bytes.

Dynamic Memory Allocation

The MallocMem, FreeMem, and CollectMem services allow allocating and freeing of small blocks of memory on demand.

The dynamic memory allocation services allocate memory from a dynamic memory pool. Memory that is freed is returned to the dynamic memory pool. Memory is allocated only while it is being used.

If there is not enough memory available in the dynamic memory pool, a new page is taken from the Memory Page Pool and added to the dynamic memory pool. When all allocations from this memory page are freed, the page is returned to the memory page pool.

The services do not keep owner, size, and validity information on the memory allocations. Therefore, when a process terminates, memory not freed is not returned to the dynamic memory pool. Memory must be explicitly freed using FreeMem before the process terminates.

The MallocMem and FreeMem services can be called from within software handlers or interrupt handlers.

The CollectMem service can be used to force the return of free pages in the dynamic memory pool to the memory page pool. For performance reasons, the kernel may not always automatically return a free page to the memory page pool.

Data Cache

The firmware supports use of the i960 processor data cache on ARTIC960 adapters that have data cache enabling hardware.

The Status Utility Configuration message shows whether data cache hardware is present on the adapter. The `DATA_CACHE` kernel configuration parameter enables use of data cache if the hardware is present. Refer to the *ARTIC960 Programmer's Reference* for information on the Status Utility and kernel parameters.

The following memory areas can be cached:

- A stack section of a process
- A data section of a process
- Memory created by a process

The loader allows the stack and/or data section of a process to be designated as cacheable. The CreateMem and MallocMem services allow a process to designate created memory as cacheable.

The i960 data cache does not support bus snooping. Therefore, memory that can be accessed by the system bus or by the daughter card Interface Chip cannot be cached. Any attempt to access cached memory by either the system bus or daughter card results in cache coherency problems. Only memory regions that are accessed only by the 80960 should be cached.

If the use of data cache is enabled, the ARTIC960 kernel creates its internal data structures as being able to be cached. Additionally, the kernel can be loaded with its stack and/or data sections designated as being able to be cached.

Big-Endian Memory Addressing

The ARTIC960 Support for OS/2, Version 1.2.1, ARTIC960 Support for AIX, Version 1.2 or higher, and ARTIC960 Support for Windows NT, Version 1.0 or higher, support use of big-endian memory addressing on ARTIC960 adapters that have big-endian enabling hardware. On these adapters, the `MEM_BIG_ENDIAN` option can be used with the memory allocation services to provide access to memory in which data is stored in the big-endian format. Normally, all ARTIC960 memory data is stored in little-endian format.

Big-endian addressing can be used to facilitate shared memory communications between the ARTIC960 and systems which are big endian by default. For example, system memory on the RISC System/6000 is stored in the big-endian format. Data can be transferred directly between the system unit and the adapter without having to perform byte swapping when big-endian memory regions are used.

Internal Data RAM

The ARTIC960 Support for OS/2, Version 1.2.1, supports the use of i960 internal data RAM. However, the kernel does not currently manage this data. Refer to the *ARTIC960 Programmer's Reference* for information on using internal data RAM.

Summary

The Memory Allocation/Suballocation Services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
CreateMem	Allocates a new block of memory. Takes parameters for sharing, access privileges, and boundary alignment.
OpenMem	Gets access to a previously-allocated block of memory. Takes parameters for access privileges.
CloseMem	Releases access to a block of memory. When the last process closes the memory, the block is returned to the free pool.
ResizeMem	Returns a portion of previously-allocated memory.
SetMemProt	Changes the access rights for the calling process to a block of memory.
SetProcMemProt	For device drivers or subsystems, this service changes the access rights for the calling process to a block of memory.
QueryMemProt	Queries the access rights for a calling process to a block of memory.
QueryProcMemProt	For device drivers or subsystems, this service queries the access rights for a process to a block of memory.
QueryFreeMem	Queries the total amount and size of largest contiguous block of free memory.
InitSuballoc	Initializes a block of memory for suballocation. Takes parameters for suballocation unit size, suballocation unit alignment, and suballocation pool size.
GetSuballoc	Suballocates memory from a suballocation pool.
FreeSuballoc	Returns suballocated memory to a suballocation pool.
GetSuballocSize	Returns the size of memory that should be allocated to make a suballocation pool with given suballocation unit size and alignment. This service should be called before CreateMem to find out how much memory should be allocated for a suballocation pool.
MallocMem	Allocates memory from the dynamic memory pool.
FreeMem	Returns memory to the dynamic memory pool.
CollectMem	Returns unused pages from the dynamic memory pool to the main memory page pool.

Process Synchronization

Process synchronization is accomplished using semaphores and events. Each allows a process to notify another process that some action has occurred. Detailed descriptions of semaphore and event services follow. For a list of the services, refer to page 27. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each.

Semaphores

Semaphores are the post/wait mechanism for all processes. A semaphore can exist as one of two types: mutual exclusion or counting. (The type is defined when the semaphore is created.)

- A **mutual exclusion (mutex)** semaphore is used for serializing access to code or data structures. It can take on a count of 0 or 1. A count of 0 indicates that it is in use, whereas a count of 1 means that it is available.
- A **counting semaphore** is used for signaling between processes or for maintaining a count of a resource—for example, the number of free buffers in a buffer pool. It can assume a count ranging from 0 to 32767. Any count other than 0 means that it is available.

Extra error checking is performed for mutex semaphores, such as not letting the same process request a mutex semaphore twice in a row without releasing it in between and not letting a process release a semaphore it does not own. Additionally, if an owner closes a mutex semaphore that it owns, or if it is stopped while it owns a mutex semaphore, all processes waiting on the semaphore are awakened with an error and the mutex semaphore count is re-initialized to 1 (available).

A process obtains a semaphore by performing a RequestSem. If a semaphore is available when requested, its count is decremented and the requesting process continues to run. If not, the requesting process is blocked. Issuing a ReleaseSem causes a semaphore to be released. When a semaphore is released, a process blocked on that semaphore is made ready to run. (If it is a higher priority than the running process, it runs immediately.) When more than one process is blocked on the semaphore, the first process that is blocked is the first process to be made ready to run. When no processes are waiting, the semaphore count is incremented.

Semaphores can be explicitly allocated and manipulated by processes by way of the calls described in this section. These semaphores are defined as *explicit*. Semaphores are also allocated by the kernel for use with queues, mailboxes, and other resources. These semaphores are defined as *implicit*. The kernel manages the requesting and releasing of these implicit semaphores for the process. Implicit semaphores cannot be used directly by semaphore services. They can be used in the event services that follow.

Using Mutex Semaphores

A mutex semaphore protects resources from simultaneous access by multiple processes. For example, there may be a critical section of code that is used by more than one process but cannot be reentered. If, at the beginning of this critical section, it performs a RequestSem and at the end it does a ReleaseSem, it will be protected. For example, if Process A calls a routine that has a critical section but is preempted by Process B that then calls the same routine, Process B blocks until Process A finishes the critical section.

Using Counting Semaphores

A counting semaphore provides a signaling mechanism between two processes or a subsystem/device driver and its interrupt handler. A process can use this type of semaphore to block until a specific action has occurred.

As an example, assume Process A needs to be notified when Process B has completed some activity. Process A creates a counting semaphore, setting its initial count to 0. Process B opens this semaphore. Process A then requests the semaphore. Because its count was 0, Process A blocks on the semaphore. Later, when Process B has completed its activity, it releases the semaphore. This causes Process A to be placed on the dispatch queue to run. If Process A is a higher priority than Process B, A preempts B and runs immediately. If A's priority is equal to or lower than B's priority, B runs until it either blocks itself or is preempted by a higher priority process.

Because counting semaphores maintain a count of the number of times they have been released, they can indicate the number of times an action has occurred. In the preceding example, if Process B was equal to or had a higher priority than Process A, Process B could perform its action *n* times before Process A ran. In this case, the semaphore count would be *n*. Process A could call RequestSem *n* times before it would again be blocked.

A further application of counting semaphores is to use them to keep a count of the number of resources available. For example, if Process A is sharing five instances of a resource with Process B, the semaphore count could be set to 5. Prior to obtaining an instance of the resource, each process would perform a RequestSem and, after releasing the resource, it would do a ReleaseSem. In this way, if all five of the resource instances were in use at the same time, the requesting process would block until a resource became available.

Events

Processes can wait for up to 32 semaphores with event services. The process builds a list of semaphore handles and control information describing whether the process should wait for any or all of the semaphores. The list of semaphore handles can be any combination of implicit and explicit semaphores, as well as any combination of mutual exclusion and counting semaphores. If a process is waiting for an explicit semaphore as part of an event, the semaphore is decremented before control is returned to the process. If a process is waiting for an implicit semaphore as part of an event, the semaphore count is not decremented before control is returned to the process. The semaphore is decremented when the process calls the service (such as queues and mailboxes) that uses the semaphore.

Using Events

Typically, events are used by processes to wait for one in a group of semaphores to become available. For example, Process A may have an implicit semaphore named ONE associated with a mailbox, and semaphore TWO associated with an action from Process B. Process A needs to wake when it receives a message in its mailbox or when Process B signals it. (See *Mailboxes* on page 28 for more information on mailboxes.) To accomplish this, Process A creates an event with the handles of the two semaphores. It then does a WaitEvent using the `EVENT_WAIT_ANY` option. Process A is now blocked until the semaphore count of either of the two semaphores goes non-zero. When Process A wakes up, it receives a status indicating which of the two semaphores caused it to run. If it awoke as a result of semaphore TWO becoming available, it must get the message from its mailbox prior to performing another WaitEvent. Otherwise, it immediately returns because there is still a message in the mailbox.

If the same semaphore is being used in an event wait and direct wait through RequestSem, these rules must be followed:

- If the semaphore is released, processes waiting for the event take priority over processes waiting with RequestSem. If the event is satisfied, the process waiting for the event is awakened before a process waiting on just the semaphore. This more evenly balances who gets the semaphore, since events are harder to satisfy.
- If a process calls RequestSem for the semaphore where processes are currently waiting for the event, but the semaphore is available, the caller of RequestSem gets ownership of the semaphore because the processes waiting on the event still have not gotten the other semaphores that make up the event.

Summary

The Process Synchronization Services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
CreateSem	Allocates a new semaphore.
OpenSem	Gets access to a previously-allocated semaphore.
CloseSem	Gives up access to a semaphore. When the last process closes a semaphore, the semaphore ceases to exist.
ReleaseSem	Makes a semaphore available to the next process waiting on it. If no processes are waiting on the semaphore, its count is incremented.
RequestSem	Waits on a semaphore until it is available. If the semaphore count is positive, the count is decremented.
QuerySemCount	Returns the current count of a semaphore.
SetSemCount	Sets the initial count of a semaphore.
CreateEvent	Allocates a new event.
OpenEvent	Gets access to a previously-allocated event. The process must already have access to the event's semaphores.
CloseEvent	Releases access to an event. When the last process closes the event, the event ceases to exist.
WaitEvent	Waits for a list of semaphores. The service takes a mask as a parameter to specify which semaphores should be included in the wait. This allows processes to create one event with all their semaphores, and then wait for any subset of the semaphores.

Process Communication

Process communication can be accomplished through queues, mailboxes, and signals. The services defined for each are listed on page 33. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each.

Queues are the most primitive of the three methods and offer the least protection from corruption by an ill-behaved application. They require that shared memory exist between processes that use queues. Further, the queue element linkages reside within the queue element itself. Other than the queue element linkages, the format of the queue element is up to the application.

Mailboxes are the preferred method of interprocess communications. Messages sent to a mailbox can reside either in process-shared or private memory. Because mailbox queue linkages are maintained separately from the mailbox message storage, mailbox message format is totally left to the application.

Mailbox messages can be sent between processes on the same ARTIC960 adapter, different adapters, and an adapter and the system unit. (Messages cannot reside in shared memory when they travel off the adapter.)

Signals are provided as a party-line method that processes can use to send messages to other processes. Signals are similar to software interrupts in that a signal can be sent from one process to one or more processes, and the receiving process does not have to be dispatched to receive it. The message contained in the signal does not have to be in shared storage.

Queues

Queues allow processes to communicate by way of shared memory. Therefore, when queues are used, the storage for the queue elements must come from memory that is accessible by all processes using the queue. Queues also allow a process to communicate with its interrupt handler. Because an interrupt handler shares memory with its owner process, shared memory is the default in this case.

Queue elements are added and removed from a queue through the PutQueue and the GetQueue services. Queue elements can be put on the queue in first in first out (FIFO) or last in first out (LIFO) order. Because the order is specified when the element is placed on the queue, a high-priority queue element can easily be placed at the top of a FIFO queue. GetQueue optionally removes the element from the queue or leaves the element in place on the queue.

A specific queue element on a queue can be located through the SearchQueue service. This service locates a queue element based on either its address or the value of a location within the queue element.

Because the queue linkages are contained within the queue element, the size of the queue element must be two words larger than the size needed for the application.

A common application for queues is as a message-passing mechanism between a device driver process time and its interrupt handler.

Mailboxes

Mailboxes allow processes to send messages to one another. Mailboxes are similar to queues but provide additional function. For example, mailbox messages can be sent not only between processes on the same adapter but between other adapters and the system unit.

A mailbox connection is one directional. To receive messages, a process creates a mailbox. Processes wanting to send messages to that mailbox must open it. Therefore, for two processes to each send messages to each other, each must create its own mailbox and open the other’s mailbox.

When a mailbox is created, storage for messages that arrive at that mailbox is also allocated—based on the parameters specified in the CreateMbx service. Other processes

can now *open* the created mailbox, providing they know the mailbox name. The opener determines whether it shares the message storage area with the mailbox creator or has its own pool. Sharing the storage area is the most efficient means of passing messages because only a pointer to the message must be passed, rather than an actual copy. However, copying the message provides more isolation between processes. Messages sent between adapters and system unit as well as adapter to adapter are automatically copied.

At create time, a mailbox is defined as being *local* or *global.* A global mailbox can receive messages from processes located on another adapter or the system unit. A local mailbox can only receive messages originating from processes on its adapter.

At open time, the caller can specify where to look for the mailbox to be opened. A global search looks on other adapters or the system unit if the mailbox is not found on the opener's adapter. A local search looks only on the opener's adapter.

The GetMbxBuffer and FreeMbxBuffer services provide a means to get and free mailbox buffers. Mailbox messages must be placed in a mailbox buffer prior to sending.

SendMbx accepts a pointer to a message buffer to send to another mailbox. If the message buffer pools are not shared, the SendMbx copies the message from one mailbox pool to another. Otherwise, it just passes a pointer to the message to the destination mailbox. It accepts an option to automatically return the message buffer to the free pool if the sender and receiver are not sharing message buffer pools. (It is not returned when sharing pools because the message could be lost.) When pools are shared, it also optionally copies the message, if requested.

ReceiveMbx returns a pointer to the next available message in the mailbox. It accepts an optional timeout to wait for a message to appear. The message also can be read from the mailbox without being removed.

Refer to the *ARTIC960 Programmer's Reference* for a list of System Unit mailbox restrictions and more information on system-unit mailbox services and ARTIC960 kernel mailbox API services.

The concept of message buffer-pool sharing provides much flexibility for optimizing the performance of mailboxes used on the same adapter. More than one mailbox can be created using the same message buffer pool. Thus, processes can receive messages in one or more of their mailboxes and pass them along to mailboxes of other processes without ever performing a copy of the message. *Table 4-1 on page 30* gives a complete picture of the memory-sharing options available and the method of specifying each option. The column entitled "Local Mailboxes" deals with mailboxes where the creator and opener are on the same unit (not supported in the system unit). The "Remote Mailbox" column is for mailboxes where the creator is on one unit and the openers are on another. Each delineates how the mailbox memory names passed on the create/open calls are used to achieve a given option.

Table 4-1. Mailbox Memory Options

Memory pool configuration	Local mailboxes	Remote mailboxes
Creator and openers of the same mailbox do not share memory pools	• Creator specifies a named memory pool or a null-name memory pool • Openers specify a named memory pool different than the creator's or a null-name memory pool	• Creator specifies a named memory pool or a null-name memory pool. • Openers specify a named memory pool or a null-name memory pool. Because the openers and creator are on different units, the storage pool names can be the same and the storage is not shared.
Creator and openers of the same mailbox share memory pools	• Creator specifies a named memory pool • Openers specify the same named memory pool as the creator	Not valid for remote mailboxes
Openers of the same mailbox (but not the creator) share memory pools	• Creator specifies a named or null-named memory pool • Openers specify the same named memory pool but different than the creators (if specified)	When openers are on same unit, they specify the same named memory pool. Invalid when openers are on different units.
Creators of different mailboxes (but not the opener) share memory pools	• Creators specify the same memory area name • Openers specify a memory area name different than the creators' name or a null memory name	When creators are on same unit, they specify the same named memory pool. Invalid when creators are on different units.
Openers of different mailboxes (but not the creator) share memory pools	• Creators specify different named memory pools or null memory area names • Openers specify the same named memory pool but different than the creators' (if specified)	Same as local for openers on the same unit. Invalid across units.
Creators and openers of different mailboxes share memory pools	• Creators specify the same memory pool • Openers specify the same named memory pool as the creators	Same as local for creators and openers on the same unit. Invalid across units.

In summary, creators and openers of the same or different mailboxes on the same unit share a memory pool only if they specify the same memory-pool name. As with any resource, NULL named pools cannot be shared. Also, pools cannot be shared across adapters or the adapters and the system unit.

Using Mailboxes

The following flows describe sending a mailbox message between process A and process B, using mailboxes that share memory and mailboxes that do not.

Shared Memory

- Process A does a CreateMbx with a mailbox name of “Steve” and a memory name of “Louise.” It receives a mailbox handle.
- Process B does an OpenMbx with a mailbox name of “Steve” and a memory name of “Louise.” It receives a mailbox handle.
- Process B does a GetMbxBuffer using Steve’s handle and receives a pointer to a buffer.
- Process B puts the message in the buffer and does a SendMbx with the buffer pointer as the message pointer parameter.
- Process A does a ReceiveMbx. It receives a pointer to the message buffer that contains the message.
- Process A processes the message and then does a FreeMbxBuffer with the buffer pointer as a parameter.

Private Memory

- Process A does a CreateMbx with a mailbox name of “Steve” and a memory name of “Louise.” It receives a mailbox handle.
- Process B does an OpenMbx with a mailbox name of “Steve” and a memory name of “Chris.” (When not on the same adapter, the name could have been “Louise.”) It receives a mailbox handle.
- Process B does a GetMbxBuffer using Steve’s handle and receives a pointer to a buffer.
- Process B puts the message in the buffer and does a SendMbx with the buffer pointer as the message pointer parameter.
- Process B either does a FreeMbxBuffer for the buffer pointer at this point, or if correlation with a response is required, it waits until the response is received to free the buffer.
- Process A does a ReceiveMbx. It receives a pointer to the message buffer that contains the message.
- Process A processes the message and then does a FreeMbxBuffer with the buffer pointer as a parameter.

Signals

Signals are modeled after a hardware bus. There is always one sender and there can be more than one receiver. Selective addressing of receivers as well as broadcast modes are supported.

CreateSig and OpenSig allow the caller to define and access a signal. They specify an optional entry-point address to be called when the signal is called. (If the entry point is not specified, the signal can only be sent and not received). Additionally, the caller can specify that it only wants to receive the signal when it is sent with a given signal ID.

The InvokeSig call sends a signal. It accepts a pointer to the information to be sent. The sender can optionally request that all processes that have access to the signal be notified, regardless of signal ID.

Typically, signals are used to send a short message from one process to a group of processes. Processing done in signals is kept to a minimum because they are a software equivalent of hardware interrupts and no processes can run while a signal is being performed. While signals can be used as an interface mechanism between a server process and its clients, it is best for the server process to declare itself as a device driver or subsystem.

Summary

The process communication services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for detailed descriptions.

Service	Description
CreateQueue	Allocates a new queue.
OpenQueue	Gets access to a previously-allocated queue.
CloseQueue	Releases access to a queue. When the last process closes the queue, the queue ceases to exist.
PutQueue	Adds an element to a queue. If processes are waiting for a queue element, the first process is awakened and given the new queue element.
GetQueue	Gets or peeks at the top element of a queue. If the queue is empty, the process is blocked, with an optional timeout, until an element is put on the queue.
SearchQueue	Searches the queue for an element with a matching address or key value. The service optionally removes the element from the queue.
CreateMbx	Allocates a new mailbox. Takes parameters for message unit size and number of message units.
OpenMbx	Gets access to a previously-allocated mailbox.
CloseMbx	Releases access to a mailbox.
GetMbxBuffer	Allocates a mailbox buffer. If no buffers are available, the process can wait optionally until a buffer is available.
FreeMbxBuffer	Returns a mailbox buffer to the free pool. This is typically called by a process that receives a mailbox message.
SendMbx	Sends a mailbox message. The message buffer is acquired with GetMbxBuffer.
ReceiveMbx	Receives a mailbox message. The process can wait with a timeout if the mailbox is empty.
CreateSig	Creates a new signal, and optionally registers a signal handler.
OpenSig	Gets access to a previously-created signal, and optionally registers a signal handler.
CloseSig	Releases access to a signal. This service also de-registers the signal handler of a process, if necessary. When the last process closes the signal, the signal ceases to exist.
InvokeSig	Calls the registered signal handlers and passes them a parameter block. Signal handlers can be called selectively or a broadcast can be done.

Timer Support

Timer support includes software timers, time-of-day clock, and a performance timer. Page 35 lists the services defined for each. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each.

Software Timers

Software timers provide processes with a fairly accurate method of measuring a time period. Time periods can range from 5 milliseconds to approximately 65 seconds with a granularity of 5 milliseconds.

Software timers come into existence by calling CreateSwTimer. Unlike most other resources, a software timer cannot be shared; therefore, there is no need for an open software timer service.

The StartSwTimer service starts the timer running. It takes a time value in milliseconds ranging from 1 to 65535 (values are be rounded up to a multiple of 5), the address of a timer handler that is called when the timer expires, and whether the timer should restart itself after it expires. The caller can also pass a *TimerMemo* value. The kernel does not examine the contents of this parameter but passes its value to the timer handler when the timer expires. It is useful as an identifier when a timer is used for more than one purpose.

A timer handler is similar to an interrupt handler. Although it does not actually run on an interrupt level, it should limit the amount of time it executes. Hardware interrupts can interrupt software handlers, but no process can run until a timer handler has completed.

Timer handlers can use any of the kernel services available to interrupt handlers. Refer to the *ARTIC960 Programmer's Reference* for a list of services callable from an interrupt handler.

Because software timers are multiplexed off a single hardware timer, some degree of inaccuracy can be expected. If a high degree of accuracy is required, special timer hardware must be added through a daughter card.

Time of Day

The time-of-day clock gives applications easy access to time-of-day information. Applications not needing this capability can disable it through the `TIME_OF_DAY` parameter in the kernel configuration file. It can be initialized when the kernel's base device driver is loaded (or anytime thereafter) by setting the -T option of the application loader (ricload). Refer to the *ARTIC960 Programmer's Reference* for more information.

Time of day can also be set by an application using the SetSystemTime call, as explained in the *ARTIC960 Programmer's Reference*. Once set, time can be read by calling QuerySystemTime. If synchronization with the system unit or another adapter is required, it must be provided by the application.

Performance Timer

ARTIC960 provides a performance timer for use by developers to obtain precision-timing information. It has a range of 1 microsecond to approximately 5.5 seconds. Because it is a single hardware resource and intended as a developer's tool, it does not require create or open calls. If multiple processes are using it, they must agree on a strategy for serializing access to it.

The performance timer is started using the StartPerfTimer call. If it is already running, when StartPerfTimer is issued, an error is returned. It can be read and stopped by the ReadPerfTimer and StopPerfTimer calls, respectively.

Summary

The timer services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
CreateSwTimer	Allocates a new software timer.
CloseSwTimer	Releases access to a software timer.
StartSwTimer	Starts a timer. The service takes a timeout Size, a timer handler that should be called on timer expiration, a flag specifying whether the timer should be restarted after expiration, and a memo field to be passed to the timer handler on timer expiration.
StopSwTimer	Cancels a running timer.
SetSystemTime	Sets the time-of-day clock.
QuerySystemTime	Reads the current time of day.
StartPerfTimer	Starts the performance timer.
StopPerfTimer	Stops the performance timer and returns its value.
ReadPerfTimer	Reads the performance timer without stopping it.

Hooks

The kernel provides hooks so that processes can be notified of special actions. These hooks have the option of pre-processing or post-processing. In other words, processes can be notified either before the action occurs or they can be notified after the action occurs. This notification takes the form of calling a hook handler registered by the process. Within the hook handler, the process can take whatever actions are required.

Only one hook is initially provided, which is for the dispatcher. A dispatcher hook handler might want to save and restore an environment for processes as they are dispatched.

Refer to the *ARTIC960 Programmer's Reference* for information on RegisterHook and DeregisterHook.

5 Kernel Asynchronous Events

This chapter provides information on conditions than can occur on the adapter.

Asynchronous Events Notification

Asynchronous event notification provides a means for processes on the ARTIC960 adapter to be made aware of certain hardware and software events. A process can use the RegisterAsyncHandler call to register that an asynchronous event notification handler be called when an event occurs. Notification is canceled by calling DeRegisterAsyncHandler. Refer to the *ARTIC960 Programmer's Reference* for a description of both calls.

An asynchronous event handler is similar to an interrupt handler, and its processing time should be limited. Also, some kernel services are not available. Refer to the *ARTIC960 Programmer's Reference* for a list of the modes in which each kernel service can be called.

Asynchronous events are categorized as software events, processor events, and adapter events.

- **Software Events**
  - Process stop
  - Process start
  - Device driver termination
  - Closing a shared resource
- **80960 Processor Events**
  - Arithmetic
  - Constraint
  - Operation
  - Protection
  - Type

- **Adapter Events**
  - Watchdog timeout
  - Parity (multibit ECC error and local bus parity)
  - Memory protection violation (80960 master)
  - Memory protection violation (system bus master)
  - PCI bus error.

If non-existent memory is accessed, memory-protection violations can occur, even with memory protection turned off.

The asynchronous events are also categorized according to their severity into three main groups: normal events, process error events, and adapter error events. This classification determines the action that the kernel takes after calling the registered asynchronous event notification handlers.

Normal Events

Normal events are those that are not errors. Execution resumes after all handlers have been called. The following events are classified as normal events:

- Software Events
  - Process stop
  - Process start
  - Device driver termination
  - Closing a shared resource
  - PCI bus error

Process Error Events

Process error events are errors that occur at process time and cause the currently executing process to be stopped. The following events are classified as process error events:

- 80960 Processor Events
  - Operation
  - Arithmetic
  - Constraint
  - Protection
  - Type
- Adapter Events
  - 80960 memory protection violation at process time
  - AIB bus read parity error with 80960 master

Adapter Error Events

Adapter error events are errors that always cause all processes to be stopped. These include process error events that occur while executing in software handlers. Specifically, all processes are stopped when an error occurs in an interrupt handler, timer handler, asynchronous event notification handler, or device driver/subsystem open, close, or call (invoke). The following events are classified as adapter error events:

- 80960 Processor Events occurring in software handlers
  - Operation
  - Arithmetic
  - Constraint
  - Protection
  - Type
- Adapter Events
  - Watchdog timeout
  - Parity error (multibit ECC error and local bus parity error)
  - System bus memory protection violation
  - 80960 memory protection violation
  - PCI bus error.

The PCI bus error event allows the asynchronous event handler to return a value indicating whether the event should be treated as a normal event or an adapter error event. Refer to the *ARTIC960 Programmer's Reference* for more information on RegisterAsyncHandler.

Terminal Error Notification

The system unit processes can be notified of terminal errors on the adapter. Terminal errors are defined as errors that are so severe that adapter processes cannot notify system unit processes through the standard communications channels, such as mailboxes. Terminal errors are composed of the adapter events defined in the previous section, as well as failures of the kernel or related subsystems. Processes in the system unit can be notified of terminal errors by the system unit device driver service RICGetException. Refer to the *ARTIC960 Programmer's Reference* for information on RICGetException and terminal error code definitions. The actual terminal error information is passed in a structure called RIC_Except, which has the following definition:

```
struct RIC_Except
{
   RIC_ULONG         ExceptionCode;
   RIC_ULONG         ExceptionDataSize;
   union
   {
      struct RIC_AsyncEvent    EventInfo;
      struct RIC_Invalid_Intr  InvIntr;
      struct RIC_Data_Corrupt  BadData;
      struct RIC_Kern_Init     KernIni;
      struct RIC_MBXErrInfo    MBXInfo;
      struct RIC_SCBErrInfo    SCBInfo;
      struct RIC_MCErrInfo     MCInfo;
      struct RIC_RPErrInfo     RPInfo;
      struct RIC_HxErrInfo     HxInfo;
   } ExceptionData;
};
```

where:

ExceptionCode
: Is an error code indicating which terminal error has occurred on the adapter. The operating system support needs to check this code when interrupted by an adapter. If ErrCode is 0, no error occurred. The exception codes follow.

ExceptionDataSize
: Is the size of the exception-specific data plus the ExceptionCode and ExceptionDataSize fields.

EventInfo Asynchronous event information detailed in the next section (see page *41*)

InvIntr Interrupt error information (see page *42*)

BadData Kernel failure due to internal data corruption (see page *43*)

KernIni Kernel-initialization error information (see page *43*)

MBXInfo External mailbox failure information (see page *44*)

SCBInfo SCB subsystem failure information (see page *45*)

MCInfo System bus I/O failure information (see page *45*)

RPInfo Information specific to a nonmaskable interrupt (NMI) on the ARTIC960Rx and the ARTIC960RxD adapters (see page *46*)

HxInfo Information specific to a PLX 9080 interrupt on the ARTIC960Hx adapter (see page 46)

The system unit device driver needs to understand only the first two words: Exception Code and Data Size. The rest of the parameters only needs to be copied, and is understood and formatted by the dump facility or status utility.

Exception Dependent Data Structures

The system unit device driver needs to understand only the first two words, ExceptionCode and ExceptionData. The rest of the parameters are to be copied, and will be interpreted and formatted by the Dump or Status utility.

The ExceptionData is a union of several structures. The structure used depends on the value in ExceptionCode. The table below shows the data structures associated with each code.

ExceptionCode	**ExceptionData**
TERMERR_WATCHDOG TERMERR_PARITY TERMERR_MEM_PROCESSOR TERMERR_MEM_MICROCHANNEL TERMERR_MEM_AIB TERMERR_PROCESSOR	**EventInfo** If one of these exceptions occurs, EventInfo is returned as ExceptionCode. Refer to the *ARTIC960 Programmer's Reference* for a definition of this structure.

ExceptionCode	ExceptionData

TERMERR_INVALID_INTR	**InvIntr**
	If this exception occurs, InvIntr is returned as ExceptionData, using the following format.
	`struct RIC_Invalid_Intr`
	`{`
	`RIC_ULONG VectorNum;`
	`RIC_ULONG ProcType;`
	`RIC_ULONG IntPend;`
	`union`
	`struct`
	`{`
	`RIC_ULONG NMIPend;`
	`RIC_ULONG XI7Pend;`
	`RIC_ULONG X16Pend;`
	`RIC_ULONG InbPend;`
	`struct`
	`{`
	`RIC_ULONG NMIPend;`
	`RIC_ULONG XI7Pend;`
	`RIC_ULONG X16Pend;`
	`RIC_ULONG InbPend;`
	`} RP;`
	`} Pend;`
	`}`
	where:
	VectorNum
	Is the vector number that interrupted.
	ProcType
	Is the processor type. The value can be equal to one of the following:
	PROC_960CX
	PROC_960HX
	PROC_960JX (ARTIC960Rx or ARTIC960RxD adapter)
	IntPend
	Is the value of the Interrupt Pending Register (IPND).
	If the adapter is the ARTIC960Rx or ARTIC960RxD adapter, the following information is also provided:
	`NMIPend`
	NMI Interrupt Status Register (NISR)
	`XI7Pend`
	XINT7 Interrupt Status Register (X7ISR)
	`XI6Pend`
	XINT6 Interrupt Status Register (X6ISR)
	`InbPend`
	Inbound Interrupt Status Register (IISR)

ExceptionCode	ExceptionData

TERMERR_DATA_CORRUPTION	**BadData**
	If this exception occurs, BadData is returned as ExceptionData, using the following format.
	`struct RIC_Data_Corrupt`
	`{`
	`RIC_PROCESSID ProcessId;`
	`void *ItsPCB;`
	`RIC_ULONG ResHandle;`
	`RIC_ULONG RetCode;`
	`}`
	where:
	ProcessId
	Process whose data structure was bad.
	ItsPCB
	Pointer to the internal data structure for the process. See note.
	ResHandle
	Specific resource returning a return code indicating bad data.See note.
	RetCode
	Actual return code.
	Note: If RetCode is `RC_INVALID_CALLER_POSITION`, a stack overflow has occurred. The ItsPCB field indicates the process stack address that caused the overflow, and the ResHandle field indicates the address limit for the stack.

TERMERR_KERNEL_INIT	**KernIni**
	If this exception occurs, KernIni is returned as ExceptionData, using the following format.
	`struct RIC_KernInitErr`
	`{`
	`RIC_ULONG FailureCode;`
	`}`
	where:
	FailureCode
	Is the reason for the kernel failure.

ExceptionCode	ExceptionData

TERMERR_EXTMAIL_FAIL	**MBXInfo**
	If this exception occurs, MBXInfo is returned as ExceptionData, using the following format.
	`struct RIC_MBXErrInfo`
	`{`
	`RIC_ULONG FailureCode;`
	`RIC_ULONG ErrType;`
	`}`
	where:
	FailureCode
	Is the reason the subsystem failure. Valid values are:
	`TERMERR_NO_MORE_MEM`
	There was not enough memory left in the internal pools for the operation to be performed.
	`TERMERR_NO_MORE_QUEUES`
	There was no queue available for the operation to be performed.
	`TERMERR_NO_MORE_SEM`
	There was no semaphore available for the operation to be performed.
	`TERMERR_NO_MORE_TIMERS`
	There was no timer available for the operation to be performed.
	`TERMERR_MC_ERRA`
	A system bus error occurred while an attempt was being made to read a mailbox message.
	`TERMERR_INVOKING_RIC_MCIO`
	An error occurred during an attempt to access the System Bus Subsystem.
	`TERMERR_INVOKING_RIC_SCB`
	An error occurred during an attempt to access the SCB Subsystem.
	ErrType
	Type of error that occurred. If the failure code is `TERMERR_MC_ERR`, possible values are:
	`TERMERR_DATA_PARITY`
	`TERMERR_CHCK`
	`TERMERR_CARD_SEL_FDBACK`
	`TERMERR_LOSS_OF_CHANNEL`
	`TERMERR_LOCAL_BUS_PARITY`
	`TERMERR_EXCEPTION`
	`TERMERR_TIMEOUT`
	If the failure code is `TERMERR_INVOKING_RIC_MCIO` or `TERMERR_INVOKING_RIC_SCB`, ErrType is the kernel return code. Otherwise, this field is not used.

ExceptionCode	ExceptionData

TERMERR_SCB_FAIL	**SCBInfo**
	If this exception occurs, SCBInfo is returned as ExceptionData, using the following format.
	`struct RIC_SCBErrInfo`
	`{`
	`RIC_ULONG FailureCode;`
	`RIC_ULONG McErrType;`
	`}`
	where:
	FailureCode
	Is the reason the subsystem failure. Valid values are:
	`TERMERR_NO_MORE_MEM`
	There was not enough memory left in the internal pools for the operation to be performed.
	`TERMERR_MC_ERRA`
	A system bus error occurred during an attempt to enqueue an SCB control element.
	MC*ErrType*
	Is the type of error that occurred if the failure code is `TERMERR_MC_ERR`. Otherwise, this field is unused. Valid values are:
	`TERMERR_PIPE_ACCESS`
	`TERMERR_PIPE_TIMEOUT`

TERM_MC_IO_SYSFAIL	**MCInfo**
	If this exception occurs, MCInfo is returned as ExceptionData, using the following format.
	`struct RIC_MCErrInfo`
	`{`
	`RIC_ULONG FailureCode;`
	`RIC_ULONG ProcessID;`
	`}`
	where:
	FailureCode
	Is the reason the subsystem failure. Valid values are:
	`TERMERR_NO_MORE_MEM`
	Not enough memory is left in the internal pools for the operation to be performed.
	ProcessID
	Is the Process ID of the process that issued the system bus operation.

ExceptionCode	ExceptionData

TERMERR_NMI_INTERRUPT	**RPInfo**
	If this exception occurs, RPInfo is returned as ExceptionData, using the following format.
	`struct RIC_RPErrInfo`
	`{`
	`RIC_ULONG NMIISR;`
	`RIC_ULONG PBISR;`
	`RIC_ULONG SBISR;`
	`RIC_ULONG PATUISR;`
	`RIC_ULONG SATUISR;`
	`RIC_ULONG COREISR;`
	`RIC_ULONG MEAR;`
	`RIC_ULONG IISR;`
	`}`
	`where:`
	NMIISR
	NMI Interrupt Status Register (NISR)
	PBISR
	Primary Bridge Interrupt Status Register (PBISR)
	SBISR
	Secondary Bridge Interrupt Status Register (SBISR)
	PATUISR
	Primary ATU Interrupt Status Register (PATUISR)
	SATUISR
	Secondary ATU Interrupt Status Register (SATUISR)
	COREISR
	Local Processor Interrupt Status Register (LPISR)
	MEAR
	Memory Error Address Register (MEAR)
	ISSR
	Inbound Interrupt Status Register (IISR)

TERMERR_PLX_INTERRUPT	**HxInfo**
	If this exception occurs, HxInfo is returned as ExceptionData, using the following format.
	`struct RIC_HxErrInfo`
	`{`
	`RIC_ULONG PLXStatus;`
	`}`
	where:
	PLXStatus
	Status portion of the PLX 9080 Status/Command Register.

TERMERR_ASYNC_NO_MORE_RES	If this exception occurs, there is not enough memory in the internal asynchronous event pools to process the asynchronous event. Therefore, the event cannot be processed.

6 Kernel Trace Services

Use Kernel Trace Services to capture information about adapter activity.

Trace APIs

The ARTIC960 kernel provides APIs that allow adapter processes to trace kernel calls or paths, or both, through their own code. Utilities are provided to dump and display this data. (See *Trace Utilities* on page 58 for more information.) Page *48* lists the kernel trace services available. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Trace information is stored in a trace buffer whose size is set by the InitTrace service. Also set is whether tracing should cease when the trace buffer is full or whether trace information should wrap.

Tracing can be selectively enabled and disabled on a service class basis by the EnableTrace and DisableTrace calls. Examples of kernel service classes are MAILBOX_SERVICE and SEMAPHORE_SERVICE. Applications can define their own service classes. 128 unused service classes are available.

Users can place their own trace information in the trace buffer by calling LogTrace. It accepts a pointer to user-defined data to place in the trace buffer as well as a format indicator. The format indicator tells the trace utilities how to format the trace buffer for display.

printf C Function

The **printf** C function can also be used to capture data on the adapter. This function works with the kernel trace services. The ARTIC960 C support library allows processes running on the adapter to use the **printf C** function to write data to stdout or stderr. The trace buffer used by the LogTrace service serves as the output device. The data can be recovered in the same way as the LogTrace data by using the system unit trace utilities. Refer to the *ARTIC960 Programmer's Reference* for information.

Calling the **printf C** function automatically enables tracing if it has not been enabled previously. A 1 KB buffer is allocated for accumulating the printf data. Whenever the printf buffer is full, the 1 KB of data is written to the LogTrace buffer using service class C_CLIB. The procedure IDs used are `P_FD_STDOUT` or `P_FD_STDERR`.

The printf buffer is written automatically to the LogTrace buffer when a process exits. The **fflush** function can be used to force a write of the printf buffer before it is full.

Summary

The kernel trace services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
Init Trace	Initializes a trace buffer for logging
EnableTrace	Enables one or more service classes for tracing
DisableTrace	Disables one or more service classes for tracing
LogTrace	Places trace data in the trace buffer

7 Kernel Commands

At its initialization, the kernel establishes a mailbox of its own for receiving commands from the system unit or other adapters. These kernel commands are listed on page 52. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each command.

The name of the kernel's command mailbox is "RIC_KERNMBX*n*", where *n* is the logical adapter number. Commands sent to the kernel mailbox take the following form. For more information on logical adapter numbers, see *Base API Services* on page 54.

```
struct RIC_KernCommand
{
   struct  RIC_KernMbxCmd                    Header;
   union
   {
      struct RIC_RegisterResponseMbxCmd      Cmd0;
      struct RIC_DeregisterResponseMbxCmd    Cmd1;
      struct RIC_QueryProcessStatusCmd       Cmd2;
      struct RIC_StopProcessCmd              Cmd3;
      struct RIC_StartProcessCmd             Cmd4;
      struct RIC_UnloadProcessCmd            Cmd5;
   }Cmds;
};
struct RIC_KernMbxCmd
{
   RIC_ULONG   CommandNum;
   RIC_RESPMBX RespMbxID;
   RIC_ULONG   CorrelationID;
   RIC_ULONG   ReturnCode;
   RIC_ULONG   Reserved;
};
```

where:

CommandNum
: Command number unique to each kernel command.

RespMbxID
: ID returned on RegisterResponseMbx.

CorrelationID
: A value, which is not interpreted by the kernel and can be used by the requester to correlate command responses.

`ReturnCode`
: Reserved field (must be set to 0).

`Reserved` Must be set to 0.

Each kernel command returns response information that takes the following form.

```
struct RIC_KernResponse
{
   RIC_ULONG     CorrelationID;
   RIC_ULONG     ReturnCode;
   RIC_ULONG     Reserved;
   union
   {
      struct RIC_RegisterResponseMbxResp    Resp0;
      struct RIC_QueryProcessStatusResp     Resp1;
   }Resp;
{;
```

where:

CorrelationID
: A value passed in the command. It can be used to correlate command responses.

ReturnCode
: Return code that is returned by the kernel to indicate the completion status of the command.

Reserved
: Must be set to 0.

A process can send commands to the kernel mailbox and receive responses to those commands in any mailbox it specifies. To establish the response mailbox, the process must first open the kernel mailbox and then send a RegisterResponseMbx command using SendMbx. RegisterResponseMbx takes a mailbox name as input and returns a response mailbox ID. This response mailbox ID is provided on subsequent kernel commands and tells the kernel where to send command response information. Prior to issuing RegisterResponseMbx, the specified mailbox must have been created as a global mailbox. When a process terminates or no longer needs to send kernel commands, it should issue a DeRegisterResponseMbx command.

Using Kernel Commands

The following sequence describes how a system unit process obtains the status of a card process named “CardProc” on logical card 0 using the QueryProcessStatusCmd.

1. Create mailbox xyz with the `MBX_CREATE_GLOBAL` option.
2. Open mailbox KERNMBX0 using the `MBX_OPEN_SEARCH_GLOBAL` option.
3. Build a RegisterResponseMbx command in a buffer acquired using GetMbxBuffer (using the KERNMBX0 handle). The fields are set as follows:

   ```
   RIC_KernCommand.RIC_KernMbxCmd.CommandNum = KERN_REG_RESP_MBX
   RIC_KernCommand.RIC_KernMbxCmd.RegisterResponseMbxCmd.MbxName = "xyz"
   ```

4. Send the command to mailbox KERNMBX0 using SendMbx.
5. Wait for a response by calling ReceiveMbx for mailbox xyz.
6. Check RIC_KernResponse.ReturnCode to verify the command was completed successfully. Then save the response ID value contained in RIC_KernRespnse.RIC_RegisterResponseMbxResp.RespMbxID for use in later commands.
7. Return both the command and response buffers using FreeMbxBuffer.
8. Obtain a buffer and build the QueryProcessStatusCmd with the fields set as follows:

   ```
   RIC_KernCommand.RIC_KernMbxCmd.CommandNum = KERN_QUERY_PROC_STAT
   RIC_KernCommand.RIC_KernMbxCmd.RespMbxId = value saved in previous
   step
   RIC_KernCommand.RIC_KernMbxCmd.QueryProcessStatusCmd.ProcName =
   "CardProc"
   ```

9. Send the command to mailbox KERNMBX0 using SendMbx.
10. Wait for a response by calling ReceiveMbx for mailbox xyz.
11. Check RIC_KernResponse.ReturnCode to verify the command was completed successfully. Then obtain the desired status information from the RIC_ProcessStatusBlock structure.
12. Return the command and response buffers using FreeMbxBuffer.

In the preceding sequence, the first seven steps are set up to establish the response mailbox and do not need to be repeated unless a different response mailbox is desired or a DeRegisterResponseMbx command is issued. Also, the same buffer could be used to send both commands, if it was sized to accommodate the larger of the two commands.

Summary

The kernel commands are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each command.

Command	Description
RegisterResponseMbx	Registers a command response mailbox.
DeRegisterResponseMbx	De-registers a command response mailbox.
QueryProcessStatus	Gets the status and other related information for a process. This call can also be used to resolve a process name into a process ID.
StartProcess	Starts a stopped process.
StopProcess	Stops a started process. All resources acquired by the process are released. The memory for the process code, data, and stack is not released so that the process can be restarted later.
UnloadProcess	Unloads the process. All the resources acquired by the process are released. The memory for the process code, data, and stack is also released.

8 System Unit Support

The ARTIC960 system unit support consists of three categories of services:

- Base API services
- Mailboxes
- Utilities

Each is described in this chapter and in more detail in the *ARTIC960 Programmer's Reference*.

Implementing API Functions

System unit support API functions are implemented using the following executables and libraries.

Operating System	Executables and Libraries	Description
OS/2	**ricio16.sys**	OS2 3.0 device driver
OS/2	**ricio32.dll**	Base API dynamic link library routines
OS/2	**ricmbx32.exe**	Mailbox process
OS/2	**ricmbx32.dll**	Mailbox-process dynamic link library routines
AIX	**ricio**	AIX device driver
AIX	**libric.a**	Base API library
AIX	**ricmbx**	Mailbox daemon
AIX	**libmbx.a**	AIX mailbox API library
Windows NT	**ibma960.sys**	Windows NT 4.0 device driver
Windows NT	**librica.dll**	Base API dynamic link library routines
Windows NT	**librica.lib**	Base API dynamic link library

These files must be loaded and configured before the associated API calls are used. Refer to the *ARTIC960 Programmer's Reference* for more information.

Base API Services

The base API services allow a process to perform various low-level operations, such as reading and writing adapter memory, resetting an adapter, and getting configuration and exception information. The operations are implemented in the following.

- OS/2 — Device driver and associated dynamic link library routines
- AIX — Device driver and associated library routines
- Windows NT — Device driver and associated services

See page 55 for a list of the services. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each. There is no concept of synchronization/signaling between adapter processes and system unit processes provided in the base API. All signaling should use the mailbox facility.

Because a system unit can support more than one ARTIC960 adapter, each adapter is referred to by a logical card number.

OS/2 In OS/2, logical card numbers range from 0 to 6 and are assigned by the device driver at its initialization. The device driver scans the physical system first for a Micro Channel bus, and then for a PCI bus. Depending on the bus found, the following occurs:

Micro Channel bus
The driver checks each slot, from low slot number to high slot number, and assigns logical card numbers consecutively. These logical card numbers range from 0 to 6 (for a maximum of seven Micro Channel bus adapters).

PCI bus
The driver interrogates the PCI BIOS for all ARTIC960 PCI, ARTIC960Hx, and ARTIC960Rx adapters, in this order. For each adapter found, the device driver assigns a logical card number starting at one greater than the number of Micro Channel bus cards (for example, the first logical PCI card can be 0 to 6). The driver supports a maximum of seven PCI adapters.

There is no correspondence between the adapter logical card number and the slot it occupies. The slot number is stored as 0xFF. Slot numbers can be accessed through the ARTIC960 RICGetConfig base API.

AIX In AIX, logical card numbers are equivalent to device minor numbers assigned by the AIX configuration manager. The supported range for logical card numbers is 0 to 13.

Windows NT
In Windows NT, logical card numbers range from 0–6 and are assigned by the device driver at its initialization. The device driver scans the PCI bus for ARTIC960 PCI adapters.

Prior to calling any of the base services, the adapter must be opened by way of the RICOpen call. It takes a logical card number as input and returns a handle. The handle is used on subsequent base API calls. When the application process is finished using base API calls, it should close the adapter by calling RICClose.

A process can read or write ARTIC960 memory using the RICRead and RICWrite services. RICRead reads data at a specified ARTIC960 address and places it into the

specified system unit buffer. RICWrite writes data from a specified system unit address to a specified ARTIC960 adapter address. Lengths can be as much as 64 KB. All ARTIC960 adapter addresses are local memory addresses—not system bus addresses.

A process can reset the card to its power-on state by issuing RICReset. When a card is reset, the kernel, any application processes, and configuration information is lost.

Fatal errors on the adapter are reported to system unit processes using the RICGetException call. An application can call this service with a timeout, or it can wait until an exception occurs. RICGetException returns exception data in the form of a structure called *ExceptData*. For more information on exceptions, see *Chapter 5: Kernel Asynchronous Events* on page 37.

Summary of Base API Services

The base API services are summarized below. Refer to the *ARTIC960 Programmer's Reference* for a detailed description of each service.

Service	Description
RICOpen	Gets access to the ARTIC960 device
RICClose	Releases access to the ARTIC960 device
RICRead	Reads data from ARTIC960 memory into system unit memory
RICWrite	Writes data from system unit memory into ARTIC960 memory
RICReset	Resets a ARTIC960 adapter
RICGetConfig	Gets hardware configuration information
RICGetVersion	Gets software version numbers
RICGetException	Gets ARTIC960 adapter exception information

Mailboxes

Mailboxes on the system unit follow the same API as mailboxes on ARTIC960 adapters. For details on their operation, refer to *Mailboxes* on page 28. However, there are some restrictions on system unit mailboxes that do not apply to adapter mailboxes. The following are restrictions for both OS/2 and AIX Mailboxes:

- There is no concept of local mailboxes, that is, mailboxes for communicating between processes residing only on the system unit. Mailbox communication can occur only between system unit processes and ARTIC960 processes. Memory pool sharing between mailboxes is still possible. For example, the creator of a mailbox on the system unit can share memory with the opener of a mailbox on the system unit.
- A single allocation using GetMbxBuffer cannot exceed 65,503 bytes.

OS/2 Mailboxes

- Because OS/2 does not support counting semaphores, the event semaphore returned on CreateMbx does not reflect the number of messages in the mailbox. It has only two states. It is cleared when messages are available and set when the mailbox is empty. A process wanting to perform a semaphore wait using the mailbox semaphore must first call ReceiveMbx with the no-wait timeout before waiting for the semaphore.

AIX Mailboxes

- Superuser authority is required to call the Mailbox Daemon.
- The `MBX_PIN_MEMORY` option is ignored in AIX CreateMbx and OpenMbx.
- The Mailbox API registers the SIGALRM signal during the first call to CreateMbx or OpenMbx. Therefore, mailbox users must not use AIX functions that require the SIGARLM signal or use the SIGARLM signal in any way.
- Mailboxes are supported only for logical cards 0 through 9.
- A child is *forked* for each mailbox application. This child is dormant until the mailbox application exits. Its function is to notify the mailbox daemon that the mailbox application died and the child exits.
- When using a ARTIC960 PCI adapter, applications on both the system unit and the card can issue SendMbx requests if the `SIZE` parameter is in the range $0 < \text{size} \leq 16384$.

Windows NT Mailboxes

Not supported.

Utilities

The utilities for the ARTIC960 adapter follow.

Task	Utilities
Initialize the ARTIC960 operating environment	Loader, Reset, Configuration
Debug and performance	Status, Dump, Trace
Run hardware diagnostics	Diagnostics

Each is described briefly in the following paragraphs and in more detail in the *ARTIC960 Programmer's Reference*.

Application Loader

The loader utility accepts relocatable executable files in the *Common Object File Format* supplied by the Intel 80960 linker. It performs relocatable addressing as required and downloads the file to the specified logical card number. The loader is used to load the kernel and its subsystems, as well as application processes.

Refer to the *ARTIC960 Programmer's Reference* for a list of the loader options. Options include the capability to perform the following.

- Specify the file name of a list of files to be loaded.
- Pass parameters to the loading process in the form of C language *argc,argv,* either on the command line or through a separate file.
- Instruct the loader to wait until the loading process executes the kernel CompleteInit service. Waiting allows a process to return initialization information for display by the Application Loader.
- Start a previously-loaded process that has not yet been started.
- Unload a previously-loaded process.

Reset Utility

The reset utility allows a user to reset a card to its power-on state. When a card is reset, the kernel and any application processes and configuration information are lost. Multiple adapters can be reset with a single call of the reset utility. Refer to the *ARTIC960 Programmer's Reference* for more detailed information on this utility.

Configuration Utility

The Configuration utility configures the adapter for communication between itself and the system unit or other adapters, or both. It must be run after the kernel and its subsystems are loaded, but prior to application loading. If a card reset is performed, the utility must be rerun.

This utility specifies the size of pipes that are used for off-card communications. It defaults to a size that should accommodate most applications. However, if necessary, the pipe size may be explicitly specified when the utility is called.

Refer to the *ARTIC960 Programmer's Reference* for more detailed information on this utility.

Status Utility

The Status utility is a debug tool that allows a developer to display information about the state of the adapter and the processes loaded on it. Some of the information provided are:

- Slot number, I/O address, memory size, and so forth
- List of processes loaded and their states
- Memory display
- List of all resources allocated
- List of resources allocated by a process
- Details about each resource allocated
- Exception conditions
- Vital Product Data (VPD) information for the adapter and the daughter card, if a daughter card is connected

This utility also is used to display memory and register information gathered by the Dump utility.

Refer to the *ARTIC960 Programmer's Reference* for more detailed information on this utility.

Dump Utility

The Dump utility captures an image of ARTIC960 memory and stores it to disk for later display by the Status utility. It has two modes of operation: triggered and immediate. When running in triggered mode, the utility waits until an adapter exception condition before dumping memory. In immediate mode, the utility performs its function when called.

Refer to the *ARTIC960 Programmer's Reference* for more detailed information on this utility.

Trace Utilities

The ARTIC960 kernel provides APIs that allow adapter processes to trace kernel calls and paths through their own code (for more information on kernel trace services see *Chapter 6: Kernel Trace Services* on page 47). There are three system unit utilities that aid in gathering and displaying trace data from the adapter: Set Trace, Get Trace, and Format Trace.

Set Trace Initializes, enables, and disables tracing of specified services.

Get Trace Reads the trace buffer from the adapter and stores it on the system unit in a user-definable trace file.

Format Trace
Formats the trace file into a user-readable format.

Refer to the *ARTIC960 Programmer's Reference* for more detailed information on these utilities.

Diagnostics Utility

The Diagnostics utility is used for running hardware diagnostics. It is installed with the operating system support programs. Refer to the installation readme file provided with the support programs for information on using this utility.

9 Compiling and Linking Programs

This chapter explains how to compile and link:

- ARTIC960 programs
- OS/2 system unit programs
- AIX system unit programs for AIX Version 4

ARTIC960 Programs

ARTIC960 adapter-resident programs are compiled and linked using the Intel set of 80960 processor C language tools. These tools include a compiler, linker, disassembler, and librarian.

All programs using ARTIC960 kernel services must include the file **ric.h**. Prior to your `#include` statement, the constant `RIC_KERNEL` must be defined to obtain the proper declarations. The following code fragment illustrates this.

```
#define RIC_KERNEL
#include <ric.h>
```

To compile the program **test.c**, call the compiler as follows:

```
ic960 -c -ACA -Gbc test.c -I <path where header files are
installed>
```

The libraries for kernel services are contained in the file **libricc.a** for OS/2 and **libriccx.a** for AIX. To link test.o with the kernel and other standard C libraries:

```
lnk960 <path where library is installed>ricproc.ld test.o -r\
-o test.rel -ACA -L<path where library is installed>
```

After linking in AIX, the relocatable executable file must be converted to little endian by using the CTOOLS960 utility, cof960. To convert **test.rel** from big endian to little endian, call cof960 as follows:

```
cof960  -lv test.rel
```

Processor Architecture Considerations

The architecture compiler option (-A) is used to specify the target instruction set.

If a module is loaded on a card which has a processor architecture that is different from the one specified when the module is compiled, the loader may display a warning message indicating processor mismatch. The module may be using instructions that are not supported on the target processor.

Using the architecture compiler option -ACA produces code that will run on all ARTIC960 adapters.

OS/2 System Unit Programs

ARTIC960 system unit-resident programs are compiled and linked using the IBM C Set/2 language tools. These tools include a compiler, linker, source level debugger, and make facility.

All programs using ARTIC960 system unit services must include the file **ric.h**. Prior to your `#include` statement, the constant `RIC_OS2_32` must be defined to obtain the proper declarations. The following code fragment illustrates:

```
#define RIC_OS2_32
#include <ric.h>
```

To compile the program **test.c**, call the compiler as follows:

```
icc /C /Gt test.c
```

The libraries for base API calls are in the **ricos232.lib** file. The libraries for mailbox calls are in the **ricmbx32.lib** file.

To link the test program **test.obj** with the base API and mailbox calls, call the linker as follows:

```
link386 /NOI test.obj,test.exe,,ricos232.lib ricmbx32.lib;.
```

The IBM VisualAge C ++ for OS/2 language tools also can be used.

AIX System Unit Programs

ARTIC960 system unit resident programs are compiled using the C for AIX Compiler.

All programs using ARTIC960 system unit services must include the file: **ric.h.** Prior to your `#include` statement, the constant `RIC_AIX_RS6000` must be defined to obtain the proper declarations. The following code fragment is an example.

```
#define RIC_AIX_RS6000
#include <ric.h>
```

To compile the program **test.c**, call the compiler as follows:

```
cc -o test -I/usr/lpp/devices.artic960/include test.c
```

The libraries for base API requests are in the **libric.a** file; the libraries for mailbox calls are in the **libmbx.a** file.

To link the test program test.o with the base API and mailbox calls, call the linker as follows:

```
cc -o test  -L/usr/lpp/devices.artic960/bin \
            -lric -lmbx test.o
```

Windows NT System Unit Programs

ARTIC960 system unit resident programs are compiled using the Microsoft 32-bit C/C++ Optimizing Compiler for 80x86.

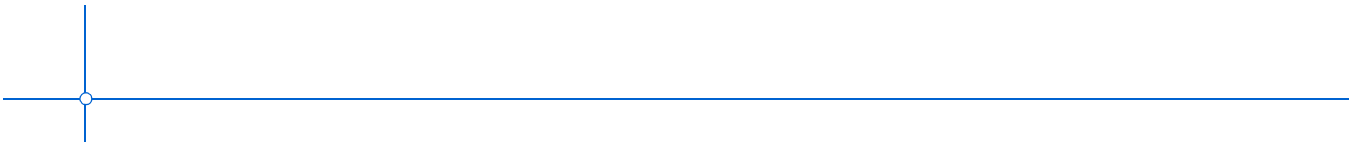
All programs using the ARTIC960 system unit services must include the file **ric.h**. Prior to your `#include` statement, the constant `RIC_WINNT` must be defined to obtain the proper declarations. The following code fragment is an example:

```
#define RIC_WINNT
#include <ric.h>
```

To compile and link the program **test.c,** call the compiler as follows:

```
cl test.c -Ic:\ric\inc -DRIC_WINNT /link c:\ric\lib\librica.lib
```

The libraries for the base API requests are in the **librica.lib** file.



Index

Numerics

80960 processor events *37*

A

access
- device driver/subsystem *12*
- rights, constants *20*
- rights, resource *19*

adapter events *38*
adapters, supported (chart) *2*
address
- ARTIC960 adapter *55*
- daughter card DMA *19*
- entry-point *32*
- I/O *57*
- memory *15*
- parameter block *14*
- queue element *28*
- timer handler *34*

addressing, big-endian memory *23*
AIX system unit programs *60*
allocate
- memory *18*
- resource *18*
- resource memory *21*

AllocHW — Allocate an Interrupt Vector *11*
AllocVector — Allocate an Interrupt Vector *11*, *13*
APIs, base services *54*
application loader description *56*
architecture considerations, processor *59*
ARTIC960 programs *59*
ASCII string name *17*
asynchronous events
- description *37*
- notification *37*
- terminal error notification *40*

B

banks, memory *18*
base API services *54*
big-endian memory *20*, *23*
block
- calculate size *21*
- initialize *12*, *24*

books, reference *ix*
broadcast modes *32*
buffer-pool sharing *29*
bus, Micro Channel/PCI *54*

C

cache, i960 processor data *22*
cached memory areas *22*
call
- create *17*
- open *17*
- signal handlers *32*

call/close entry points *15*
child, mailbox *56*
CloseDev — Close a Subsystem or Device Driver *13*
commands, kernel
- summary *52*
- using *51*

compile and link
- ARTIC960 programs *59*
- OS/2 system unit programs *60*
- Windows NT system unit programs *61*

CompleteInit — Mark Process as Completely Initialized *7*, *12*
constants, set of *19*
conventions, notational *viii*
counting semaphores, using *25*
CPU access *19*
create
- mailbox *29*
- memory *20*
- process *7*

create call *17*
CreateDev — Register a Subsystem or Device Driver *11*
CreateProcess — Create a Process *7*
creator, mailbox *26*
critical code section *25*

D

data cache, enable *22*
data formatting/transmission *1*

data/code pointers 14
DATA_CACHE parameter 22
daughter card/system bus access 19
debug tool 57
depth count 5
device driver
  accessing 12
  description 11
  initialization 11
  memory-protection maps 14
  services 15
device driver/subsystem
  memory protection 14
  ric_base.rel, file 2
diagnostics utility 58
DMA channels 18
dump utility, description 57
dynamic memory
  allocation 15, 22
  management 18

E

e-mail address, RadiSys ix
enable data cache 22
EnterCritSec — Enter Critical Section 5
entry point, call/close 15
error
  checking, mutex semaphores 25
  events, adapter 39
  events, process 38
  notification, terminal 40
  PCI bus 38
event
  asynchronous 37
  notification, asynchronous 37
  processes/semaphores, wait 26
  using 26
exception conditions 37
exception data 55
ExitCritSec — Exit Critical Section 5
explicit semaphores 25

F

fatal errors 55
fflush C function 47
FIFO (first in first out) 28
forked child 56

G

global mailbox 29
global memory option 14

H

handler
  interrupt 13
  timer 34
hardware
  diagnostics 58
  features 1
  resources 18
high-priority interrupt (trap) 8
hooks, overview 35
HXInfo 46

I

image, relocatable 2
implicit semaphores 25
initialization
  device driver/subsystem 11
  process 6
input/output subsystem 2
instance data services, process 7
instruction
  fetches 19
  memory access 18
internal data RAM 23
interprocess communications 27
interrupt handlers 13, 34
InvokeDev — Call a Subsystem or Device Driver 13

K

kernel
  asynchronous events 37
  description 2
  device driver/subsystem 11
  hooks 35
  memory management 18
  overview 2
  process management 5
  process synchronization 24
  resources 17
  ric_kern.rel, file 2
  timer support 34
  trace services 47

L

level, priority 5
libraries
  for base API requests 60
  for kernel services 59
  lnk960 (link) 59
  ricproc.ld (link) 59
LIFO (last in first out) 28
linker, call 60

little-endian memory 20
lnk960 (link libraries) 59
load application, description 56
loader utility, application 56
local mailbox 29
logical card numbers 54

M

mailbox
  child 56
  description 28
  general information 55
  local/global 29
  memory options 18
  messages 26
  queue linkages 27
  restrictions 55
  using 31
management
  process, description of 5
  resource 17
maps, memory protection 14
mask as parameter 27
memory
  access-right constants 20
  addressing, big-endian 23
  allocation, resource 21
  cached 22
  create 20
  management 18
  open 20
  pool 18
  private 31
  protection 8, 14
  protection maps 14
  protection, process 8
  read/write 55
  resource management 18
  shared 31
  sharing, resource 20
  suballocation 21
  type 18
memory management, summary of services 24
MEMORY_PROTECTION parameters 8
message, mailbox 26
Micro Channel bus 54
mutex semaphores 25

N

name
  ASCII string 17
  process 5
NMI (non-maskable interrupt) 41
normal events 38
notational conventions viii
notification
  asynchronous event 37
  terminal error 40
null-name memory pool 30

O

On-card STREAMS Environment 3
open
  call 17
  memory 20
OpenDev — Open a Subsystem or Device Driver 12
OpenMem — Get Addressability to Allocated Memory 8
overview
  ARTIC960 1
  hooks 35
  process communication 27

P

Packet Memory access 18
parameter, DATA_CACHE 22
parameter, MEMORY_PROTECTION 8
parity error 39
PCI bus 54
peer process, defined 18
performance timer, description 35
pools
  dynamic memory 22
  message buffer 29
printf C function 47
priority level/process 5
private memory 31
process
  initialization 6
  instance data services 7
  memory protection 8
  scheduling 5
  spawning 7
  states 6
  termination 7
process communication
  description 27
  summary of services 33
process management
  overview 5
  summary of services 9
process synchronization
  overview 24
  summary of services 27
processes, number supported 5
processor

architecture considerations 59
data cache 22
processor events 37
programs
AIX system unit 60
ARTIC960 59
OS/2 system unit 60
Windows NT system unit 61
protection options, memory 14
protocol conversion 1
publications, reference ix

Q

QueryHW — Query Status of Hardware Device 11
QueryPriority — Query the Priority of the Process 5
QueryProcessInExec — Get ID of Process in Execution 5
QueryProcMemProt — Query a Process' Memory Protection 11
queue
described 28
element linkage 27

R

RadiSys, contacting ix
RAM, internal data 23
random access memory (RAM) 2
read/write adapter memory 54
reference publications ix
request semaphore 25
reset
card to power-on state 55
multiple adapters 57
utility 57
resources
access rights 19
allocation 18
dynamic memory 22
handle 17
hardware 18
keep count of 26
management 17
software 17
types of 17
ReturnHW — Return a Hardware Device 11
ReturnVector — Return an Interrupt Vector 11
ric_base.rel, kernel file 2
ric_kern.rel, kernel file 2
ric_mcio.rel, system bus I/O subsystem file 2
ric_scb.rel, SCB subsystem file 2
RICLOAD application loader 7
ricproc.ld 59
rights, resource access 19
routine called within routine 5
RPInfo 46

S

scheduling, process 5
semaphore
counting 25
described 25
explicit 25
implicit 25
mutual exclusion (mutex) 25
using mutex 25
service providers 11
set
access rights 19
depth count 5
SetExitRoutine — Set the Exit Routine for the Process 7
SetPriority — Set the Priority of the Process 5
SetProcMemProt — Change a Process' Memory Protection 11
SetVector — Set a New Interrupt Vector Entry Point 11
share resource memory 20
shared memory 31
shared resources 17
signal
description 28, 32
handlers, call 33
ID 32
software events 37
spawning, process 7
stack, overflow 43
StartProcess — Start a Process 7
states, process 6
status utility, description 57
StopProcess — Stop a Process 7
STREAMS environment 3
string, name 17
suballocation
memory 18
resource memory 21
subsystem, initialization 11
summary
base API services 55
device driver/subsystem services 15
kernel commands 52
kernel trace services 48
memory management 24
process communication services 33
process synchronization services 27
timer services 35
supported adapters 2
synchronization, process 24
system unit

APIs *54*
mailboxes *55*
programs *60*
support *1*, *53*

T

terminal error notification *40*
termination, process *7*
timers
performance *35*
services *35*
software *34*
support *34*
time of day *34*
time-slice *5*
time-slice timer *5*
tools, C language *59*
trap (high-priority interrupt) *8*
types, memory *18*

U

UNIX STREAMS *3*
UnloadProcess — Unload a Process *7*
unused service classes *47*
URL, RadiSys *ix*
utility
application loader *56*
configuration *57*
diagnostics *58*
dump *57*
general information *56*
reset *57*
RICLOAD *7*
status *57*
trace *58*

V

value
correlation (DevMemo) *12*
location in queue element *28*
TimerMemo *34*
vector sharing *14*
violation, memory protection *38*, *39*
VPD (Vital Product Data) *57*

W

watchdog timeout *39*
Windows NT system unit programs *61*
World Wide Web, accessing RadiSys *ix*