# RadiSys ARTIC960 STREAMS

# Environment Reference

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
(503) 615-1150
www.radisys.com
June 1999

Before using this information and the product it supports, be sure to read all the information in *Appendix A, Notices* .

This edition replaces and makes obsolete the previous edition.

This edition applies to the following RadiSys support programs and to all subsequent versions and releases until otherwise indicated in new editions.

- RadiSys ARTIC960 Support for OS/2, Version 1.2.1
- RadiSys ARTIC960 Support for AIX, Version 1.3.1
- RadiSys ARTIC960 Support for Windows NT, Version 1.0

These programs support the following adapter cards:

- RadiSys ARTIC960 Micro Channel
- RadiSys ARTIC960 PCI
- RadiSys ARTIC960Rx PCI
- RadiSys ARTIC960Hx PCI
- RadiSys ARTIC960RxD PCI

June 1999

# About this Guide

This book provides information on the On-card STREAMS Environment. This book does not include sample code.

## Guide contents

The following lists the contents of this Guide.

| Chapter | | Description |
|---|---|---|
| 1 | RadiSys ARTIC960 STREAMS Overview | Provides an overview for the RadiSys ARTIC960 On-card STREAMS environment. |
| 2 | AIX STREAMS960 Application Device Driver | Describes how to change or list parameters of the STREAMS S96ADD or its devices, how to enable or disable STREAMS, and also lists the supported S96ADD APIs. |
| 3 | On-card STREAMS Subsystem and Cross Bus Driver | Provides information about loading and configuring the On-card STREAMS Subsystem (OSS) and cross-bus driver (ESS). |
| 4 | STREAMS-based Module/ Driver Information | Describes how to build an on-card STREAMS-based module/driver and information about the Standard Kernel Functions (SKFs) and On-card STREAMS Subsystem (OSS) Kernel Functions (OKF). |
| 5 | Developing a Cross-bus Driver | Describes the process to develop a cross-bus driver. |
| 6 | STREAMS Access Library | Describes the STREAMS Access Library (SAL) and how requirements for using with various operating systems. |
| 7 | STREAMS Access Library Functions | Describes the memory and stream functions that are included as part of the SAL system unit support. |

### Appendices

The appendices provide additional information about RadiSys products.

| Appendix | | Description |
|---|---|---|
| A | Notices | Lists notices related to availability of RadiSys products and contact information for license information. |

# Conventions

## Notations

This manual uses the following notational conventions:

- All numbers are decimal unless otherwise stated.
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless otherwise stated.
- `Data structures and syntax strings appear in this font.`

Notes indicate important information about the product.

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

The globe indicates a World Wide Web address.

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

ESD cautions indicate situations that may cause damage to hardware via electro-static discharge.

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

## Terms

This manual uses the following terms:

*System bus:* refers to either the Micro Channel or PCI bus.

*RadiSys ARTIC960*

refers to programs that run on the RadiSys ARTIC960, RadiSys ARTIC960 PCI, RadiSys ARTIC960Rx PCI, or RadiSys ARTIC960Hx PCI adapters, or the adapters themselves.

*RadiSys ARTIC960 PCI*

refers to functions supported only on the RadiSys ARTIC960 PCI adapter.

*RadiSys ARTIC960 MCA*

refers to functions supported only on the RadiSys ARTIC960 Micro Channel adapter.

*RadiSys ARTIC960Rx PCI*

refers to functions supported by the RadiSys ARTIC960Rx PCI adapter.

*ARTIC960Hx PCI*

refers to functions supported by the RadiSys ARTIC960Hx PCI adapter.

*ARTIC960RxD PCI*

refers to functions supported by the base card of the RadiSys ARTIC960RxD Quad Digital Trunk PCI adapter.

## Symbols

This manual uses the following symbols:

- All counts in this book are assumed to start at zero.
- 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.

## RadiSys ARTIC960 Developer's Kit—Contents

The Developer's Kit is a set of publications and programs designed to help RadiSys ARTIC960 software developers develop for the RadiSys ARTIC960 platform. The following items make up the Developer's Kit:

- *RadiSys ARTIC960 Hardware Technical Reference* presents technical details of the adapter's system, options, and hardware interfaces. It provides descriptions and data related to the card configuration, functions, hardware interfaces, and programming considerations.
- *RadiSys ARTIC960 Programmer's Guide* contains information about the RadiSys 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 *RadiSys ARTIC960 Programmer's Reference* provides an overview of both the adapter kernel support and the associated processes and utilities, as well as each of the services provided by the system unit support and the adapter kernel support.
- *RadiSys ARTIC960 Application Interface Board Developer's Guide* provides the hardware and the software developer with AIB design requirements, and a collection of productivity tools to aid in the development of an AIB.
- A set of operating system packages, each containing sample programs and utilities to support the development of system unit and adapter applications. These packages are to be used with the *RadiSys ARTIC960 Programmer's Guide*, the *RadiSys ARTIC960. Programmer's Reference*, and the *RadiSys ARTIC960 Application Interface Board Developer's Guide*.
- You can obtain these books from the World Wide Web (WWW) at:

`http://www.radisys.com/products/artic/`

If you do not have access to the WWW, you can obtain these books from the no-fee Developer's Assistance Program (DAP).

## Developer's Assistance Program

In addition to the *Developer's Kit*, further programming and hardware development assistance is provided by the RadiSys ARTIC960 Developer's Assistance Program (DAP). The DAP provides, by way of 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*** RadiSys ARTIC960 DAP membership by contacting us.

By telephone, call (561) 454-3200.

By E-mail, send to artic@radisys.com.

## Where to Get More Information

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

- *RadiSys ARTIC960 Programmer's Guide*
- *RadiSys ARTIC960 Programmer's Reference*
- *IBM Operating System/2* (OS/2) Version 3.0, Advanced Interactive Executive (AIX) Version 4.1 and 4.2
- Operating and Installation documentation provided with your computer system
- *Guide to Operations* books for one of the following co-processor adapters:

  RadiSys ARTIC960 Micro Channel adapter

  RadiSys ARTIC960 PCI adapter

  ARTIC960Rx PCI adapter

  ARTIC960Hx PCI adapter

  ARTIC960RxD PCI adapter

  Each book contains a description of the co-processor adapter, instructions for physically installing the adapter, and parts listings.
- *AIX Version 4.x Kernel Extensions and Device Support, Programming Concepts* (SC23-2207)
- *RadiSys ARTIC960 Programmer's Guide and Reference*
- *XL C Language Reference* (SC09-1260)
- *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*
- For information about writing a STREAMS module or driver, refer to the AIX Web site:

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

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

# Contents

## Chapter 6: STREAMS Access Library

## Chapter 7: STREAMS Access Library Functions

# Figures

# Tables

# RadiSys ARTIC960 STREAMS Overview

# 1

This book provides information on the RadiSys ARTIC960 On-card STREAMS environment (hereafter called RadiSys ARTIC960 STREAMS). Before using this information, you must be familiar with the UNIX† STREAMS architecture.

The RadiSys ARTIC960 runtime environment provides the standard UNIX System V STREAMS Releases 3 and 4 tool set for running STREAMS-based module/drivers on an RadiSys ARTIC960 adapter. Benefits associated with RadiSys ARTIC960 STREAMS on an RadiSys ARTIC960 adapter are as follows:

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

To provide streams access and services to system unit applications, RadiSys ARTIC960 STREAMS consists of four major parts:

### STREAMS960 Application Device Driver (S960ADD)

A device driver that allows AIX STREAMS applications to communicate with a STREAMS module or driver on the adapter.

### STREAMS Access Library (SAL)

A system unit component that provides the access to the On-card STREAMS Subsystem through a system device driver application program interface (API) from both UNIX- and non-UNIX-based operating systems.

Programming to the SAL interface is needed only if you cannot use the AIX S960ADD.

See Chapter 6, STREAMS Access Library , for more information.

### On-card STREAMS Subsystem (OSS)

An on-card component that provides the UNIX System V STREAMS Releases 3 and 4 tool set on an RadiSys ARTIC960 adapter.

See Chapter 3, On-card STREAMS Subsystem and Cross Bus Driver , for more information.

### RadiSys On-card STREAMS Cross-Bus Driver (ESS)

An on-card component that provides the support to transmit STREAMS data across the system bus between SAL and OSS. See Chapter 3, On-card STREAMS Subsystem and Cross Bus Driver  for information on how to load this separately loaded module.

If you need other cross-bus support, see Chapter 4, STREAMS-based Module/ Driver Information , for instructions on how to write a cross-bus driver.

Figure 1-1 illustrates how these different components interact with each other between the system unit and an RadiSys ARTIC960 adapter.

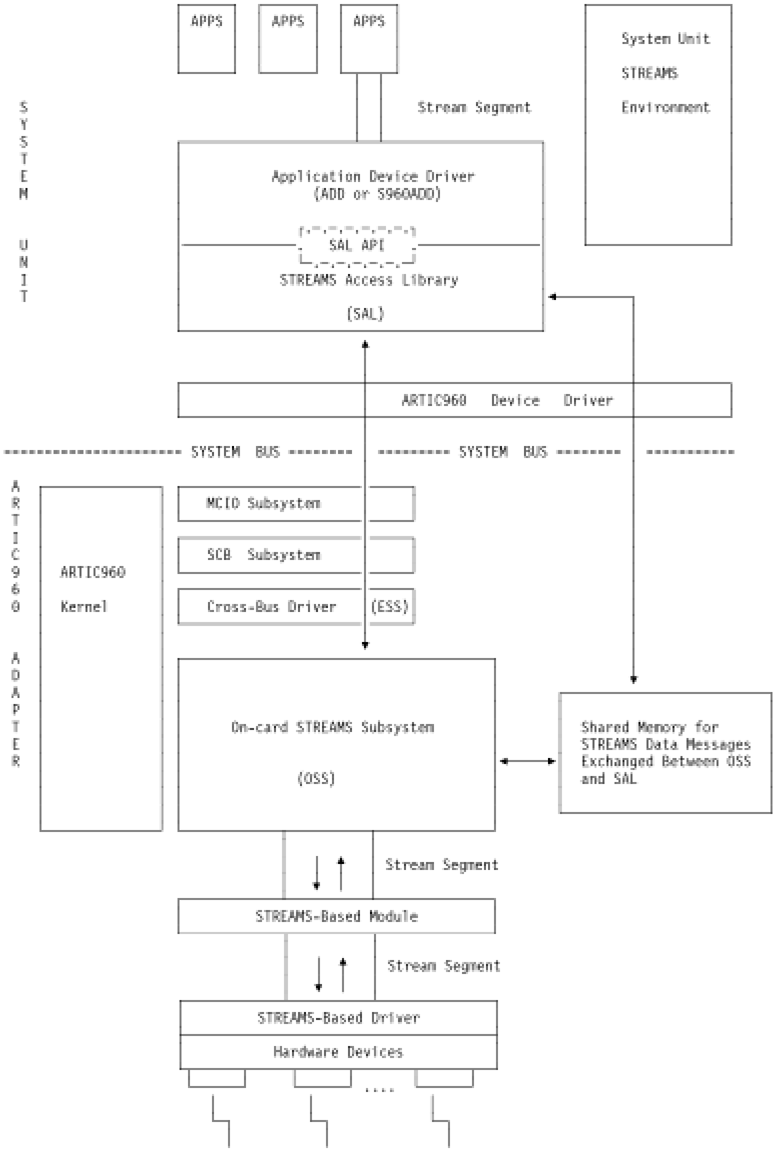

**Figure 1-1. System Unit and Co-Processor Components**

# AIX STREAMS960 Application Device Driver

# 2

The STREAMS960 application device driver (S960ADD) allows AIX STREAMS applications to communicate with a STREAMS module or driver on the adapter. With the S960ADD and OSS, STREAMS modules on the AIX system unit can put messages (putmsg) and get messages (getmsg) from STREAMS modules or drivers running on the RadiSys ARTIC960 adapter.

If you are using S960ADD, you do not need to develop your own cross-bus device driver or program to the STREAMS Access Library (SAL).

## Configuration

You can change or list parameters of the STREAMS S96ADD or its devices.

## STREAMS S960ADD

To change or list parameters to configure the S96ADD, do one of the following:

- Use the STREAMS-specific `chdev` command
- Use the SMIT menu **Change/Show Characteristics of RadiSys RadiSys ARTIC960 STREAMS Device Driver.**

### Using the Command Line

Issue the command:

```
chdev –l 'ric96add0' <-a CardPoolSize=xxx> <-a CardPacketSize=YYY>
                     <-a SalPipeTimeout=T> <-a SalMaxUpstrLen=LLL>
```

### Parameters

*CardPoolSize*

This is the number of buffers allocated. The size of each buffer is defined by CardPacketSize. The total amount of memory used on the adapter is (CardPacketSize * CardPoolSize) bytes. The default size is 200 buffers.

*CardPacketSize*

This is the size of a buffer allocated on an adapter. S960ADD and the cross-bus driver use these buffers when transferring a message to the adapter. The default size is 512 bytes.

*CardPacketSize*

This is the size of a buffer allocated on an adapter. S960ADD and the cross-bus driver use these buffers when transferring a message to the adapter. The default size is 512 bytes.

*SalPipeTimeout*
This is the value in seconds and defines the timeout value used by SAL in case of pipe full condition. The default is 5 seconds.

*SalMaxUpstrLen*
This is the maximum size of a message sent upstream from the adapter to ADD. The default size is 4106 bytes.

### Using the SMIT Menu

To display the STREAMS S960ADD configuration information:

1. Type: `smitty`
2. Select **Devices**.
3. Select **Communication**.
4. Select **RadiSys ARTIC960 STREAMS Device Driver**.
5. Select **Change/Show Characteristics of RadiSys ARTIC960 STREAMS Device Driver**.
6. Select **ric960add0**.
7. Increase or decrease the parameters as needed. Then press Enter to change.

## STREAMS S960ADD Devices

To change or list parameters to configure the STREAMS S96ADD devices, do one of the following:

- Use the STREAMS-specific cst960dev command
- Use the SMIT menu **Change/Show Characteristics of Devices of RadiSys ARTIC960 STREAMS Device Driver**.

### Using the Command Line

Issue the command:

```
cst960dev -p <HiWat|LoWat> &lbrk.-v<value>&rbrk
```

The command device parameters follow:

*High Water Mark (HiWat)*
This is the high water mark for STREAMS queues flowing from the adapter. This value is the number of bytes contained in a queue before flow control begins to block messages from being added to the STREAMS queue. The default is 0x1000 bytes.

*Low Water Mark (LoWat)*
This is the low water mark for STREAMS queues flowing from the adapter. When the value of bytes remaining in a queue reaches this level, the queue is unblocked by STREAMS. The default is 0x200 bytes.

### Using the SMIT Menu

To display the STREAMS S960ADD devices configuration information:

1. Type: **smitty**
2. Select **Devices**.
3. Select **Communication**.
4. Select **RadiSys ARTIC960 STREAMS Device Driver**.
5. Select **Change/Show Characteristics of Devices of RadiSys ARTIC960 STREAMS Device Driver**.
6. Increase or decrease the parameters as needed. Then press `Enter` to change.
7. Reboot the system to have the changes take effect.

## Enabling/Disabling STREAMS

When the S960ADD is installed, the device driver periodically queries all the RadiSys ARTIC960 adapters found in the system to determine when the OSS and the cross-bus driver (ESS) are downloaded to the adapter. These queries are usually harmless, but in cases where they impact the performance of non-STREAMS adapters, it might be desirable to unconfigure the STREAMS environment on the adapters.

STREAMS queries can be disabled by doing one of the following:

- Using the STREAMS-specific setaddmask command.
- Using the SMIT menu **Configure STREAMS Support for All RadiSys ARTIC960 Adapters.**

### Using the Command Line

Issue the command:

```
setaddmask -s ricioX -o <0|1>
```

where

X is the adapter number, and 0 means disable and 1 means enable in the <0|1> parameter..

If STREAMS is not configured on an adapter and a STREAMS message is sent, the application is returned an ENOCONNECT error.

### Using the SMIT Menu

To display the STREAMS S960ADD devices configuration information:

1. Type: **smitty**
2. Select **Devices**.
3. Select **Communication**.
4. Select **RadiSys ARTIC960 STREAMS Device Driver**.
5. Select **Configure STREAMS Support for All RadiSys ARTIC960 Adapters.**
6. Select **Enable/Disable STREAMS Support for an RadiSys ARTIC960 Adapter**.
7. Select the correct adapter.
8. Use the Tab key to toggle between Enable and Disable, and then press Enter to change.

## Supported S96ADD APIs

The S960ADD supports the following APIs:

open<br>
close
ioctl
getmsg
putmsg

# On-card STREAMS Subsystem and Cross Bus Driver

# 3

An on-card component called *On-card STREAMS Subsystem* (OSS) provides the UNIX V STREAMS Releases 3 and 4 tool set on an RadiSys ARTIC960 STREAMS.

An on-card component called *On-card STREAMS Cross-Bus Driver* (ESS) provides the support to transmit STREAMS data across the system bus between the STREAMS Access Library (SAL) and the OSS.

## Loading On-card STREAMS Subsystem

The OSS must be loaded onto an RadiSys ARTIC960 adapter prior to any On-card STREAMS-based module/drivers process. Use the RICLOAD system utility to load the OSS (ric_oss.rel). For information on RICLOAD, refer to the *RadiSys ARTIC960 Programmer's Reference*. The OSS runs at a privileged high priority when loaded onto an RadiSys ARTIC960 adapter.

Refer to the *RadiSys ARTIC960 Programmer's Reference* for information about loading the kernel and subsystems.

## Loading On-card STREAMS Cross-Bus Driver (ESS)

If the RadiSys ARTIC960 cross-bus support is being used (that is, you did not write a cross-bus driver and plan to use the cross-bus driver provided with the RadiSys ARTIC960 support), use the RICLOAD system utility to load ESS (ric_ess.rel) after the SCB subsystem and OSS.

## Configuring On-card STREAMS Subsystem

Configuration for the On-card STREAMS Subsystem is done through load-time parameters that can be passed on the command line or through a configuration file when using RICLOAD. These parameters take the form of keywords (representing specific parameters) followed by an equal sign (=) and their value. The encoding of these parameters follows the same rules as described in the *RadiSys ARTIC960 Programmer's Reference*. The configuration parameter values of the OSS are independent of the operating system being used..

Refer to the *RadiSys ARTIC960 Programmer's Reference* for information about configuring the kernel and subsystems. There is no specific tuning of configuration parameters for operation of the RadiSys ARTIC960 STREAMS.

Table 3-1 describes the load-time parameters. All parameters are numeric decimal.

**Table 3-1. Load-time Parameters**

| Parameter | Range | Default | Description |
|---|---|---|---|
| BUFREGION | 0–1 | 0 | This parameter determines how memory addresses for RadiSys ARTIC960 STREAMS buffers are formatted. If BUFREGION = 0 or is not specified, the memory addresses are in little endian format. If this parameter is specified as 1, memory addresses for STREAMS buffers are in the big endian format.<br>Once the buffer pool is allocated, all STREAMS data buffers addresses have the desired format. |
| EXPFACTOR | 1–(no upper limit) | 10 | Expansion factor for RadiSys ARTIC960 STREAMS resources.<br>This parameter indicates the number of units to increase a resource pool. Expansion is realized until the maximum number of resources is allotted. |
| LOWSCALE | 1–99 | 70 | Watermark (in percent) for low-priority RadiSys ARTIC960 STREAMS user-data-buffer memory allocation.<br>This parameter indicates a watermark under which the RadiSys ARTIC960 STREAMS user-data-buffer memory allocation will fail for low-priority requests. For example, using the default value, requests of low-priority memory allocations are honored as long as no more than 70% of the total RadiSys ARTIC960 STREAMS user-data-buffer memory pool is allocated (at least 30% of the pool is free). |
| MAXBLOCKLEN | 1–(no upper limit) | 4096 | This parameter indicates the maximum length for the user-data-buffer portion of a stream data block (dblk_t) allocation. This value represents the maximum value (in bytes) that an on-card STREAMS-based module/drivers can specify during an allocb Standard Kernel Function (SKF) API. This size must *not* include the data (dblk_t) or message (mblk_t) block used to construct the stream message. |
| MAXDATAB | 1–(no upper limit) | 200 | This parameter indicates the maximum number of RadiSys ARTIC960 STREAMS user-data-buffers (size determined by the MAXBLOCKLEN parameter) that can be allocated by the OSS. Initially, the pool of the RadiSys ARTIC960 STREAMS user-data-buffer memory is created with a minimum amount of buffers (specified by the EXPFACTOR parameter) and expanded as needed when requests from the on-card STREAMS-based module/drivers require new buffer allocations. Once expanded to the full maximum number specified with this parameter, the on-card STREAMS-based module/drivers program's request for memory is not satisfied until RadiSys ARTIC960 STREAMS user-data-buffer memory is released. Depending on the priority of the memory allocation request, the amount of available memory for allocation is determined by a scale factor specified by the LOWSCALE and MEDSCALE parameters. |

| Parameter | Range | Default | Description |
|---|---|---|---|
| MAXEXTB | 1–(no upper limit) | 200 | This parameter indicates the maximum number of RadiSys ARTIC960 STREAMS extended data blocks managed by the On-card STREAMS Subsystem. Initially, the RadiSys ARTIC960 STREAMS extended data blocks' pool is created with a minimum amount of blocks (specified by the EXPFACTOR parameter) and expanded as needed when on-card STREAMS-based module/driver's requests require new block allocations to take place (using the esballoc SKF API). Once expanded to the full maximum number specified with this parameter, the STREAMS-based module/driver's request for extended data blocks is not satisfied until one or more are released. Depending on the block allocation request's priority, the amount of available blocks for allocation is determined by a scale factor specified by the LOWSCALE and MEDSCALE parameters. |
| MAXHIGHB | 1–(no upper limit) | 4 | This parameter indicates the maximum number of STREAMS Access Library (SAL) data buffers (size determined by the MAXBLOCKLEN parameter) that are allocated by the OSS and available exclusively for the system unit SAL to realize high-priority RadiSys ARTIC960 STREAMS message allocation. Initially, the SAL data buffers' pool is created with the default value specified by this parameter. Therefore, there is no further pool expansion because it is allocated at its maximum. |
| MAXSCBQUEUED | 3–(no upper limit) | 300 | This parameter indicates a threshold beyond which transmission of RadiSys ARTIC960 STREAMS messages between the system unit and a given RadiSys ARTIC960 adapter, in the direction where congestion occurs, is stopped. Transmission resumes when the partner side (SAL or OSS) dequeues enough messages so that a low watermark level is reached (defaulted to one third of this parameter's value, which is currently 100). One direction being flow controlled does not prevent the other from functioning properly if its flow is clear. When the flow gets controlled at this level, *all* RadiSys ARTIC960 STREAMS opened with the RadiSys ARTIC960 adapter at that time get flow controlled for their low-priority messages, regardless of their own congested status. Reaching this threshold denotes a *major* communication or configuration problem between the system unit and the RadiSys ARTIC960 adapter. Finer tuning might be necessary according to the host's speed and/or available co-processor adapter memory. |

| Parameter | Range | Default | Description |
|---|---|---|---|
| MEDSCALE | 1–99 | 90 | Watermark (in percent) for medium priority RadiSys ARTIC960 STREAMS user-data-buffer memory allocation.<br>This parameter indicates a watermark under which RadiSys ARTIC960 STREAMS user-data-buffer memory allocation will fail for medium priority requests. For example, using the default value, medium priority memory allocations' requests are honored as long as not more than 90% of the total RadiSys ARTIC960 STREAMS user-data-buffer memory pool is allocated (at least 10% of the pool is free).<br>**High-priority Requests**<br>High-priority requests are not subject to restriction and are honored until there is no more RadiSys ARTIC960 STREAMS user-data-buffer memory available. |
| MINMSGLEN | 0–(no upper limit) | 64 | The minimum amount of bytes to take into account in a queue load when an RadiSys ARTIC960 STREAMS message is queued.<br>This parameter indicates a minimum amount of bytes that are counted in a queue load (q_count field) when a message whose length is lower gets queued. The purpose is to ensure that a stream carrying short messages does not exhaust memory resources and that flow control is activated before a large amount of messages get queued. A value of 0 disables this feature and the real accounting takes place, regardless of the length of the RadiSys ARTIC960 STREAMS message. |
| STRMS_PER_TASK | 1–65536 | 1048 | This parameter determines the maximum number of streams that can be opened by all system unit application processes. When the application process of a system unit opens a stream with a STREAMS-based driver, it allocates one file descriptor from the OSS. |
| STRSCBQUEUED | 3–(no upper limit) | 30 | This parameter indicates a threshold beyond which transmission of RadiSys ARTIC960 STREAMS messages between one system unit stream's segment and a given RadiSys ARTIC960 adapter stream's segment, in the direction where congestion occurs, is stopped. Transmission resumes when the partner stream participating in the communication dequeues enough messages so a *low watermark* level is reached (defaulted to one third of this parameter's value, which is currently 10). One direction being flow controlled does not prevent the other from functioning properly if its flow is clear. When the flow gets controlled at this level, only the stream being congested at that time gets flow controlled for its low-priority messages. See Flow Control on page 76 for more information. |

| Parameter | Range | Default | Description |
|---|---|---|---|
| SRVSLICE | 1–(no upper limit) | 8 | Maximum number of messages delivered when using getq SKF API during the same service procedure.<br>This parameter indicates a threshold above which a getq SKF API returns null (no more message) while having already delivered the specified number of messages during the time the same service procedure was run without being interrupted. Nevertheless, when this threshold is reached, the service procedure is automatically rescheduled after other service procedures have had a chance to run. The purpose of this feature is to enhance the RadiSys ARTIC960 STREAMS scheduling scheme by adding fairness among different applications' type, thus regulating RadiSys ARTIC960 STREAMS traffic.<br>This feature is in effect only for the queue in service. If a message is taken out from another queue than the one being serviced, the feature will not apply.<br>This support is transparent to STREAMS-based module/drivers, but developers should pay attention to the following note before enabling the feature.<br>**Attention**<br>Some on-card STREAMS-based module/drivers programs could be sensitive to such processing and not be able to support this feature. This is particularly true when a driver uses the null condition being returned as a trigger to update its own state and/or flush queues, and so forth. The feature should then be disabled by setting a fairly high value for this parameter, thus never reaching the threshold triggering the processing. (A reasonable value would be 0x0000ffff, for example, as it is very unlikely to have that amount of messages queued at one time in a given queue.) |

## Configuring On-card STREAMS Cross-Bus Driver

ESS is not configurable.

## Initialization Error and Exception Codes

When a failure occurs during the On-card STREAMS Subsystem initialization's phase, an error code is returned, which can be retrieved through the application loader's **–w** option. For kernel return codes, refer to the *RadiSys ARTIC960 Programmer's Reference*.

### 0xEERRRRRR

where:

*EE* Is the error code part

*RRRRRR* Is the kernel return code part (the lower six hexadecimal digits of the return codes listed in the *RadiSys ARTIC960 Programmer's Reference.*

## Initialization Errors

**Table 3-2. Error Classification**

| Error Code | Description |
|---|---|
| 0xF1 | Runtime parameter failure (OSSINIT_PARAM) |
| 0xF2 | Memory allocation failure (OSSINIT_ALLOC) |
| 0xF3 | Miscellaneous allocation failure (OSSINIT_MEMORY) |
| 0xF6 | Semaphore failure (OSSINIT_SCHEDSEM) |
| 0xF7 | Process information failure (OSSINIT_PRIORITY) |
| 0xF8 | Cross-Bus Main Program Loop (MPL) failure (OSSINIT_CXMPL) |
| 0xF9 | On-card Main Program Loop failure (OSSINIT_MPL) |
| 0xFA | OSS Statistics setup failure (OSSINIT_STAT) |
| 0xFB | Log device driver failure (OSSINIT_LOGDRVR) |
| 0xE0 | Cross-Bus service failure (ESSINIT_INITIALIZE) |
| 0xC0 | Memory manager service failure (CBMINIT_INITIALIZE) |

**Table 3-3. Initialization Errors**

| Error Code | Description |
|---|---|
| 0xF1RRRRRR | Error passing a runtime parameter. See *RadiSys ARTIC960 Programmer's Guide and Reference* for return code part. |
| 0xF2000001 | Invalid MAXBLOCKLEN value. |
| 0xF2000002 | Error registering data buffers memory pool. |
| 0xF2000003 | Error expanding data buffers memory pool. |
| 0xF2000004 | Error registering expanded buffers memory pool. |
| 0xF2000005 | Error expanding expanded buffers memory pool. |
| 0xF2000006 | Error expanding timers pool. |
| 0xF2000007 | Error registering high-priority buffers memory pool. |
| 0xF2000008 | Error expanding high-priority buffers memory pool. |
| 0xF3000000 | Task table allocation. |
| 0xF6RRRRRR | Error creating OSS scheduling semaphore. Return code from CreateSem() kernel service. |
| 0xF7RRRRRR | Error setting OSS priority. Return code from Query/SetPriority() kernel service. |
| 0xF8000001 | Error allocating cross-bus MPL's queues. |
| 0xF9000001 | Error allocating on-card MPL's queues. |
| 0xFARRRRRR | Error allocating memory for OSS statistics. Return code from CreateMem() kernel service. |
| 0xFBRRRRRR | Error installing the Log device driver. Return code from s96_devinst() OSS service. |
| 0xE0RRRRRR | Error initializing cross-bus service. |
| 0xC0RRRRRR | Error initializing Memory Manager Service. |

## Runtime Exceptions

The OSS might report several types of exceptions during runtime panic situations. These exceptions are *fatal* errors. When the exception occurs, the exception data should be extracted through the RadiSys ARTIC960 status utility using the “exception conditions” item from its main menu. The format of each of the exceptions shown in Table 3-4 follows the exception display format for the utility.

**Table 3-4. Exceptions**

| Exception Code | Exception Data (word 0) | Exception Data (word 1) |
|---|---|---|
| 0x04 (OSSERR_QELM) | Doublelinked head[2] | element[1] |
| 0x05 (OSSERR_QUERYPROCESS) | QueryProcessInExec retcode | n/a |
| 0x07 (OSSERR_PUTQUEUE) | queue_t[1] | mblk_t[1] |
| 0x09 (OSSERR_REQSCHEDSEM) | RequestSem retcode | Semaphore handle |
| 0x0A (OSSERR_RELSCHEDSEM) | ReleaseSem retcode | Semaphore handle |
| 0x0B (OSSERR_RECEIVE) | queue_t[1] | mblk_t[1] |
| 0x0D (OSSERR_MALLOC) | n/a | Size requested (in bytes) |
| 0x0E (OSSERR_BUFCALL) | Reason (EAGAIN,EINVAL) | |
| 0x0F (OSSERR_INVCODEPATH) | n/a | n/a |
| 0x10 (OSSERR_PANIC) | n/a | char string[1] |
| 0x11 (OSSERR_TIMER) | Start/StopSwTimer retcode | timeo[1] |

[1] Represents a pointer

[2] Represents the address of a pointer

# STREAMS-based Module/ Driver Information

4

This chapter contains information on how to build an on-card STREAMS-based module/ driver and information about the following available services to the STREAMS-based module/driver application processes.

- Standard Kernel Functions (SKFs)
- On-card STREAMS Subsystem (OSS) Kernel Functions (OKF)

### Restrictions

System Calls (getmsg, getpmsg, putmsg, and putpmsg) are *not* provided by the RadiSys ARTIC960 STREAMS because an on-card stream is opened, controlled, and maintained from the system unit.

Also, RadiSys ARTIC960 kernel calls that block are *not* allowed to be used by an on-card STREAMS module or driver because OSS is the dispatching kernel when loaded.

All services are usable at RadiSys ARTIC960 STREAMS service procedure time, which is the regular mode STREAMS-based module/drivers are run under. However, many *message handling* services cannot be used from an interrupt handler. This could impact any STREAMS-based module/driver's design.

Transparent Ioctl is not supported by RadiSys ARTIC960 STREAMS. Also, banding is not supported by RadiSys ARTIC960 STREAMS; therefore, services using banding parameters are not listed in the following table..

**Notes**

- A STREAMS-based module/driver is not intended to have a signal handler, an asynchronous handler, or a process exit routine. Consequently, these modes are not specified in Table 4-1 or Table 4-2.
- The Macro column specifies whether the function is provided as an inline macro.

The Interrupt Handler column specifies whether the function can be called from an interrupt handler.

## SKF Functions

All functions listed in Table 4-1 are considered Standard Kernel Functions (SKFs) when using UNIX System V STREAMS terminology. Refer to the *UNIX SVR4.2 Device Driver Reference* for information on these functions.

Table 4-1. Standard Kernel Functions (SKFs)

| Function | Macro | Interrupt Handler | Description |
|---|---|---|---|
| adjmsg | No | Yes[1] | Trims bytes in a message |
| allocb | No | Yes[1] | Allocates a message, data block, and data buffer |
| backq | No | No | Returns a pointer to the queue behind a given queue |
| bcopy | Yes | Yes | Copies a memory zone |
| bufcall | No | Yes[1] | Recovers from failure of the allocb function |
| bzero | Yes | Yes | Zeroes a memory zone |
| canput | No | No | Tests for available space in a queue |
| copyb | No | Yes[1] | Copies a message (single) |
| copymsg | No | Yes[1] | Copies a message (multiple) |
| datamsg | Yes | Yes | Tests whether the message is a data message |
| dupb | No | Yes[1] | Duplicates a message block descriptor (single) |
| dupmsg | No | Yes[1] | Duplicates a message block descriptor (multiple) |
| enableok | Yes | No | Enables a queue to be scheduled for service |
| esballoc | No | Yes[1] | Allocates a message and data block (no data buffer) |
| esbbcall | No | Yes[1] | Recovers from failure of the esballoc function |
| flushq | No | No | Flushes a queue |
| freeb | No | Yes[1] | Frees a message (single) |
| freemsg | No | Yes[1] | Frees a message (multiple) |
| getadmin | No | No | Returns a pointer to a module |
| getmid | No | No | Returns a module ID |
| getq | No | No | Gets a message from a queue |
| insq | No | No | Puts a message at a specific place in a queue |
| linkb | No | Yes | Concatenates two messages into one |
| Major | Yes | Yes | Extracts the major portion of a device number |
| makedev | Yes | Yes | Makes a device number with major/minor numbers |
| minor | Yes | Yes | Extracts the minor portion of a device number |
| msgdsize | No | Yes | Gets the number of data bytes in a message |
| noenable | Yes | No | Prevents a queue from being scheduled |
| OTHERQ | Yes | Yes | Returns the pointer to the write queue |
| pullupmsg | No | Yes[1] | Concatenates and aligns bytes in a message |
| putbq | No | No | Returns a message to the beginning of a queue |
| putctl | No | No | Passes a control message |
| putctl1 | No | No | Passes a control message with a one-byte parameter |
| putnext | Yes | No | Passes a message to the next queue |
| putq | No | No | Puts a message on a queue |
| qenable | No | Yes | Puts a queue in the scheduling ring |
| qreply | No | No | Sends a message on a stream in the reverse direction |
| qsize | No | No | Finds the number of messages on a queue |
| RD | Yes | Yes | Gets the pointer to the read queue |

| | | | |
|---|---|---|---|
| rmvb | No | Yes | Removes a message block from a message |
| rmvq | No | No | Removes a message from a queue |
| splstr | Yes | Yes | Sets the processor level to disable interrupts |
| splx | Yes | Yes | Exits a previous splstr condition |
| strlog | No | No | Generates error-logging and event-tracing messages |
| strqget | No | Yes | Obtains information about a queue |
| strqset | No | No | Changes information about a queue |
| suser | No | Yes | Informs about a user privilege |
| testb | No | Yes | Tests if given message size can be allocated |
| timeout | No | No | Schedules a function to be called after a specified interval environment |
| unbufcall | No | Yes[1] | Cancels a pending bufcall |
| unlinkb | No | Yes | Removes a message block from the head of a message |
| untimeout | No | No | Cancels a pending timeout |
| WR | Yes | Yes | Gets the pointer to the write queue |

[1] Call available from Interrupt Handlers *only* for OSS versions 1.2.0 and higher.

# OSS Kernel Functions

The RadiSys ARTIC960 STREAMS encompasses an extra set of functions for STREAMS-based module/drivers to use.  These functions are called *OSS Kernel Functions* (OKFs).

The sections following Table 4-2 contain more information about these functions.

**Table 4-2. OSS Kernel Functions (OKFs)**

| Function | Macro | Interrupt Handler | Description |
|---|---|---|---|
| qhipri | Yes | No | Sets a queue at high priority when scheduled |
| qlopri | Yes | No | Sets a queue at default priority when scheduled |
| s96_devinst | No | No | Dynamically installs a STREAMS-based module/ drivers in the OSS switch tables |

## qhipri

Sets a queue at high priority when scheduled

### Macro Prototype

```
qhipri ( queue_t       *q);
```

### Parameters

*q* Input. The queue pointer for which high scheduling priority has to be enabled.

### Remarks

This function enhances the RadiSys ARTIC960 STREAMS scheduling scheme by allowing some queues in the system to be prioritized when they are candidates for scheduling. A high-priority queue will be scheduled after all already queued high-priority queues but before any low priority ones. This function can be called during the queue open process or at any other time to switch from/to high priority to/from low priority.

# qlopri

Sets a queue at low priority when scheduled

## Macro Prototype

```
qlopri ( queue_t       *q);
```

## Parameters

*q* Input. The queue pointer for which regular scheduling priority has to be enabled.

## Remarks

This routine enhances the RadiSys ARTIC960 STREAMS scheduling algorithm by allowing some queues in the system to be prioritized when they are candidates for scheduling. A low priority queue will be scheduled after all already queued high- and low-priority queues (this is the regular mode of operation). This routine can be called during the queue open process or at any other time to switch from/to high priority to/from low priority.

## s96_devinst

Installs a STREAMS-based module/driver into the OSS device/module switch tables.

### Functional Prototype

```
int    s96_devinst      (int                    operation,
                         s96conf_t             *conf);
typedef struct s96conf
{
   char                 *name;
   struct streamtab     *stab;
   int                   flags;
   dev_t                 dev;
} s96conf_t;
```

### Parameters

*operation*

Input. The operation to perform. Two operations are available:

S96_LOAD_MOD

Installs a module in the module switch table

S96_LOAD_DEV

Installs a device in the driver switch table

**Note:** After the STREAMS-based module/driver is loaded successfully, the only way to unload it is to reset the adapter. There is no selectable unload operation available.

*conf* Input. Pointer to the configuration block. The following fields are significant.

*name* Input. Specifies the name of the extension.

For modules, this name is used during IOCTL operations.

For drivers, this name is used during SAL’s s96_open operations. See STE_OPEN — Open Stream on page 48 for more information. The maximum length for the name is FMNAMESZ + 1 (null-termination character included).

**Note:** FMNAMESZ is defined in the C language support **include** file.

*stab* Input. Points to a streamtab structure. The streamtab memory block must be allocated by the caller and must remain allocated for the duration of the STREAMS-based module/driver’s load.

*flags* Input. Specifies the style of the module or driver open routine. Acceptable values are (mutually exclusive):

S96_SVR3_OPEN

Specifies the open syntax and semantics used in UNIX System V STREAMS Release 3

S96_SVR4_OPEN
Specifies the open syntax and semantics used in UNIX System V STREAMS Release 4

*dev* Output. For a driver, contains the major number allocated by the OSS. The major number is formatted as a device number, so it can be manipulated using the major, minor, and makedev functions.

For a module, the value returned is –1.

*returned value*
Output. On successful handling of the request, a value of zero (0) is returned. An error code other than 0 indicates the error.

### Error Codes

On failure, the routine returns one of the following error codes:

| | |
|---|---|
| EEXIST | The extension specified already exists in the OSS switch tables. |
| EINVAL | A parameter contains an unacceptable value. |
| ENOMEM | Not enough RadiSys ARTIC960 memory is available. |

# Building a STREAMS-based Driver

The OSS provides C language support to develop custom STREAMS-based modules/ drivers running on an RadiSys ARTIC960 adapter. Such a module/driver will be built the same way as any other RadiSys ARTIC960 process. Refer to the *RadiSys ARTIC960 Programmer’s Reference* for information on how to build an RadiSys ARTIC960 adapter-resident program (RadiSys ARTIC960 process) for the RadiSys ARTIC960 adapter. In addition, the OSS C language support includes header files and a binary library that are used in order to make the RadiSys ARTIC9600 process a STREAMS-based module/driver for RadiSys ARTIC960 STREAMS.

## Using the Compile Command

The **osssvc.h** header file, provided with the RadiSys ARTIC960 STREAMS C support, enables the RadiSys ARTIC960 process to access the SKF APIs. This file *must* be included whenever an SKF call is made within the C language support module. All other RadiSys ARTIC960 STREAMS include files are to be included when a particular structure, prototype, or external definition is needed. The compile command must also define compile switches. The mandatory switch is S96. Cross-bus drivers must include the **cxbuser.h** file for cross-bus driver structure definitions.

Assuming the RadiSys ARTIC960 STREAMS and other RadiSys ARTIC960 header files are located in the directory referenced by the makefile variable RIC960_INCLUDE, the following compile command’s template can be used to generate an object file.

```
ic960 -c -ACA -Gbc -w1 -g -DS96 -DRIC_KERNEL -I. -I$(RIC960_INCLUDE) \
   danyboon.c -o danyboon.o
```

## Using the Linkedit commands

The C support library **libosse.a** participates in the linkedit phase of the build process and *must* be included.

Assuming the RadiSys ARTIC960 STREAMS and other RadiSys ARTIC960 runtime libraries are located in the directory referenced by the makefile variable RIC960_LIB, the following linkedit command’s template can be used to generate a STREAMS-based module/driver executable file or a cross-bus driver.

```
lnk960 -ACA -L$(RIC960_LIB) danyboon.o $(RIC960_LIB)/ricproc.ld -
losse
   -o danyboon.rel
lnk960 -r -ACA -L$(RIC960_LIB) danyboon.o $(RIC960_LIB)/ricproc.ld
-losse
   -o danyboon.rel
cof960 -lpv danyboon.rel
```

# Developing a Cross-bus Driver

# 5

A cross-bus driver is a special kind of streams driver that is linked *above* the OSS and provides communication across the system bus.

The OSS needs to have one cross-bus driver loaded on the adapter and attached to communicate with the system unit or another peer RadiSys ARTIC960 adapter. IBM provides ric_ess.rel, a cross-bus driver (ESS), to communicate with the STREAMS Access Library (SAL). However, if you need to provide a different way of communicating with the system unit and are not planning to use ESS, follow the instructions in this chapter. The OSS includes special macros and SKF functions for use by a cross-bus driver described in this chapter. See Chapter 4, STREAMS-based Module/Driver Information for other OSS functions and macros that can be used by the cross-bus driver and for instructions on building a cross-bus driver.

After a cross-bus driver is installed, it remains attached to the OSS until the card is reset (that is, it cannot be uninstalled).

The cross-bus driver is a loadable RadiSys ARTIC960 subsystem whose primary task during its initialization process is to register one cross-bus driver with the OSS and then go to sleep, running exclusively under the service-processing time of OSS. When successfully registered, it gets a handle back which it uses for all control requests it sends to the OSS subsystem.

Each cross-bus driver is identified by a callback routine and a parameter field which the driver has to provide during registration. The callback function is used by the OSS provider to communicate to the cross-bus driver when a control request is completed or an unsolicited event has to be delivered.

Figure 5-1 shows one example of a cross-bus driver communicating with the OSS and how services can be used to implement a cross-bus driver. In the diagram, the inbound and outbound cross-bus transfer represents the cross-bus driver’s interface to the system bus..

This implementation is only an example and will change depending on your requirements.

There should be a flow-control mechanism between the system unit application and the cross-bus driver.

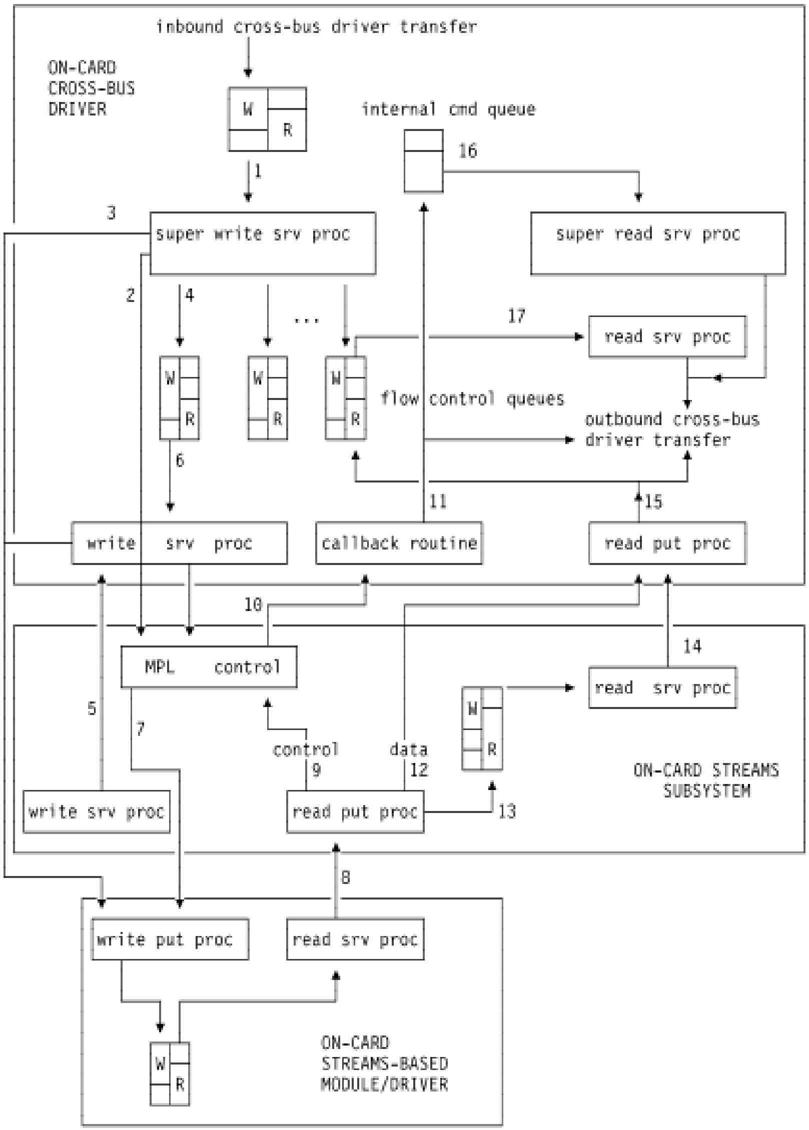

**Figure 5-1. Example of an On-card Lower-end Interface**

(1) Messages received through the cross-bus driver's upper interface are taken out from the *super* write queue and processed by the cross-bus driver's `write service` procedure (one queue for all streams).

(2) Control messages are passed directly to the Main Program Logic (MPL) control logic in OSS (using the cxb_control function).

(3) Data messages are passed directly to the first On-card STREAMS module/driver if the segment is not flow controlled (using the cxb_putnext function).

(4) Data messages are queued in the cross-bus driver's write flow-control queue if the segment is flow controlled (one queue per opened stream).

(1) Messages received through the cross-bus driver's upper interface are taken out from the *super* write queue and processed by the cross-bus driver's `write service` procedure (one queue for all streams).

(5) When the stream gets out of the flow control situation, the cross-bus driver's `write service` procedure is called, emptying (**6**) the cross-bus driver's write queue filled with elements enqueued in (**4**). Data messages are queued in the cross-bus driver's write queue if the segment goes back into flow control.

(7) The MPL control logic processes the control element given at (**2**) and, occasionally, originates a stream message for the On-card STREAMS driver/module (an I_LINK command, for example).

(8) The On-card STREAMS module/driver forwards a data message through the OSS `read put` procedure.

(9) Control messages are given to the MPL control logic which will process and eventually forward them to the cross-bus driver, as described in (**10**).

(10) The MPL control logic sends element control blocks to the cross-bus driver through the registered callback routine.

(11) The cross-bus driver's callback routine immediately performs the outbound cross-bus transfer for the element control block.

(12) Data messages are given to the cross-bus driver's `read put` procedure if not flow controlled.

(13) Data messages are queued in the OSS `read` queue if the cross-bus driver's `read queue` is flow controlled (one queue per stream).

(14) When the cross-bus driver's `read queue` becomes available, the OSS `read service` procedure gets scheduled, dequeues any data message pending in its `read` queue and forwards them to the cross-bus driver's `read put` procedure.

(15) The `read put` procedure immediately performs the outbound cross-bus transfer for the element control block or defers the transfer by queueing the block if outbound transfer is temporarily unavailable. The queueing is performed in the message queue, thus using the flow-control procedure.

(16) When the outbound cross-bus transfer becomes available again, the cross-bus driver's `super read service` procedure is run, dequeueing the internally queued control blocks first.

(17) Following (**14**), any cross-bus driver stream that was blocked because of outbound cross-bus transfer unavailability has its `read service` procedure run to dequeue and send data messages outbound.

## Element Control Block Structure

Element control blocks are exchanged during requests as well as responses/indications using the GenCXB structure.

```
struct GenCXB
{
  struct q_prefix    qp;            /* fixed OSS reserved area
*/
   unsigned long       header2[4];      /* fixed header    */
   // element control block
   // pointer to this location is passed to the control routine,
   // however the entire GenCXB block is allocated.
   union element_ctrl_block  {
     struct ipcb        oss_req;      /* for OSS requests
*/
     struct cbms_cb      cbm_req;      /* Reserved
*/
   } elmblock;
};
```

where:

*qp* Is the queue prefix. This is used by OSS only as a queueing prefix and does not need to be initialized by the cross-bus driver. However, the cross-bus driver can use the entire location to store its own data until the element control block's ownership is transferred to the OSS (using cxb_control).

*header2* Is the block header. This is reserved for use by the cross-bus driver for its own purpose. This reserved area is 16 bytes long and should never be exceeded nor reduced. There is no guarantee that the memory will not be altered during processing of the request as the GenCXB control block ownership is transferred to the OSS when the cxb_control function is called. In the same manner, the OSS will not rely on this memory location to store any data for responses/indications because it also loses ownership of the location after calling the cross-bus driver's callback function.

*elmblock*

Are the parameter blocks for various elements:

- When the callback routine's response field elmorigin is OSS_SCB_ELEM, the element control block will be formatted as an ipcb block.
- When the callback routine's response field elmorigin is CBM_SCB_ELEM, the element control block will be formatted as a cbms_cb block..

cbms_cb blocks are currently not documented or supported by cross-bus drivers.

### Callback Routine

The following explains the prototype and fields for the callback routine (cxbcallback_rtn).

```
void   cxbcallback_rtn  ( cxbresp_t         *resp);
```

```
typedef struct cxbresp
{
   unsigned long              elmorigin;
   unsigned long              cxbcallback_prm;
   unsigned long              elmblock_len;
   union element_ctrl_block  *elmblock_ptr;
   unsigned long              elmblock_memo;
   unsigned long              reserved;
} cxbresp_t;
```

where:

*elmorigin*

Is the origin of the response. This identifies the component delivering the response/indication element control block to the cross-bus driver. Currently the following values are defined:

OSS_SCB_ELEM

Originating from the OSS. This *must* be supported by the cross-bus driver interface as it conveys all stream-related elements back to the cross-bus driver.

CBM_SCB_ELEM

Originating from the CBMS. If the cross-bus driver does not use CBMS services, no CBMS elements will be delivered through this interface. Thus, this can only be *optionally* supported by the cross-bus driver interface..

The current version of this document does not include CBMS elements.

*cxbcallback_prm*

Callback routine’s parameter. This value is specified by the cross-bus driver during the CXB_REG_DRIVER operation and is provided unaltered during callback. (See CXB_REG_DRIVER for details.)

*elmblock_len*

Size (in bytes) of the element control block, including its (optional) data and control fields.

*elmblock_ptr*

Pointer to the beginning of the element control block containing the response parameters. The pointer value is set to the address of the elmblock union into the GenCXB memory block. (See page 30.) The cross-bus driver owns the GenCXB memory block while in its callback routine. It is then responsible for freeing the block (using the free() kernel service) at any time, before or after returning from the callback routine.

*elmblock_memo*

Correlation value.

**If the origin is OSS_SCB_ELEM:**

This value has been specified only once by the cross-bus driver during a STE_OPEN in the **ctltype** field of the element control block. (See page 48 for how to encode the ipcb block.)

### Device Profiles

The cross-bus driver must create at least one device profile to link its cross-bus driver queues to the stream head queues (CXB_LINK_HEAD). To create a device profile, use the s96_devinst function call as you would use it to install a STREAMS-based module/ driver. The following is mandatory when installing a device profile:

- Use the S96_LOAD_DEV operation.
- Declare a `write service` procedure in the streamtab structure.
- Provide high and low watermarks for the read side to ensure effective flow control for outbound transfers.

The cross-bus driver has no limitation on the number of profiles being defined. The device number returned by the s96_devinst function call must be saved and provided during the link head step.

## Flow Charts

The following charts show an example of how a cross-bus driver can be designed.

## Registering a Cross-Bus Driver

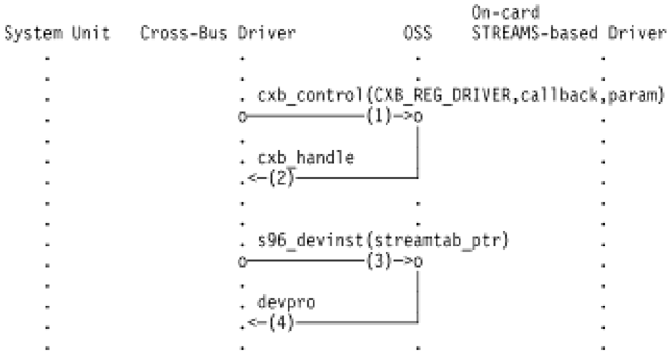
System Unit
Cross-Bus Driver
OSS
On-card STREAMS-based Driver
cxb_control(CXB_REG_DRIVER,callback,param)
(1)
cxb_handle
(2)
s96_devinst(streamtab_ptr)
(3)
devpro
(4)

**Figure 5-2. Registering a Cross-bus Driver with OSS**

(1) The cross-bus driver, during its initialization phase, registers a callback routine and a parameter using the CXB_REG_DRIVER operation.

(2) The OSS saves information into its cross-bus driver table, allocates a task ID for the new cross-bus driver and, if successful, returns a handle which is used by the cross-bus driver for its further control operations.

(3) The cross-bus driver, during its initialization phase, registers an on-card STREAMS-based module driver in order to create a profile containing addresses of service procedures, queue high and low watermarks, and so forth.

(4) The OSS saves information into its on-card STREAMS-based module/driver tables, allocates a device number and, if successful, returns this device number which is used by the cross-bus driver for its CXB_LINK_HEAD operation requiring a device profile to create the cross-bus driver queue pair.

## Linking an On-card Stream Segment

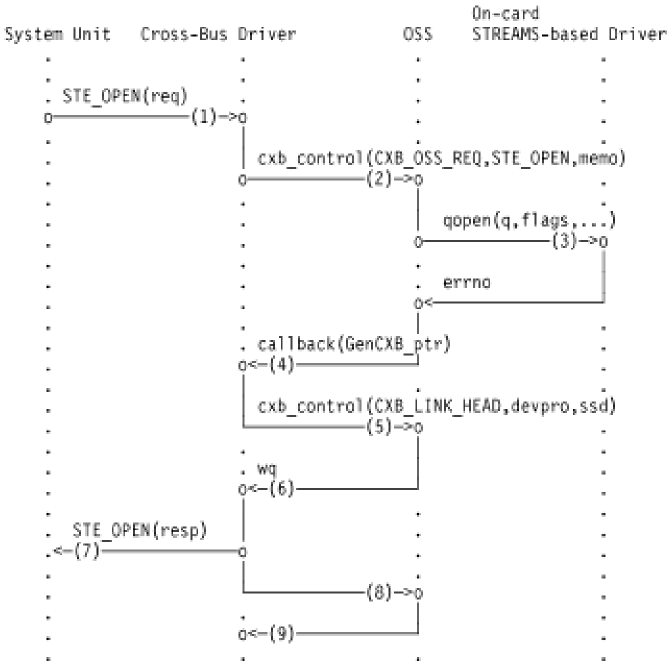

**Figure 5-3. Linking an On-card Stream Segment with the Cross-bus Driver**

(1) The system unit sends down a STE_OPEN request formatted in an ipcb block. The cross-bus driver receives the request and queues it into a STREAMS service procedure.

(2) The cross-bus driver service procedure gets control and detects that the element is a STE_OPEN control block.

- It dynamically allocates a memory block and saves the memory pointer as a memo in the reserved ctltype field of the STE_OPEN control block. (See page 48 for coding.) This pointer is used at **(7)** to initialize the *`q_ptr`* field.
- It dynamically allocates a GenCXB structure block and copies the incoming ipcb block into it.
- It performs a CXB_OSS_REQ operation, using the cross-bus driver handle acquired as described in the registration sequence in Registering a Cross-Bus Driver on page 33.

(**3**) The OSS processes the open request and calls the target driver's queue open routine. On return from the queue open routine, the return code is provided by the target driver.

(4) The OSS initiates a reply to the previous STE_OPEN request through the cross-bus driver's callback routine. The memo field saved by the cross-bus driver from the STE_OPEN request is returned during the reply.

(5) The cross-bus driver checks the open return code and, if successful, tries to link itself to the stream head queues created by OSS for the on-card stream segment by using the CXB_LINK_HEAD operation. The device profile has been acquired as described in Registering a Cross-Bus Driver on page 33.

(6) The OSS allocates a pair of queues, performs the link between the cross-bus driver queues and the stream head queues and then, if successful, returns a pointer on the write queue of the allocated cross-bus driver pair.

(7) The cross-bus driver initializes the q_ptr field of the cross-bus driver write queue with the memo value returned from the *STE_OPEN*, which was a pointer on a memory block allocated in **(2)**. This block is used by the cross-bus driver to keep instance-specific information about the on-card stream segment. Then it replies to the system unit request.

(8) The callback routine is returned.

(9) The first cxb_control function is returned.

# Unlinking an On-card Stream Segment

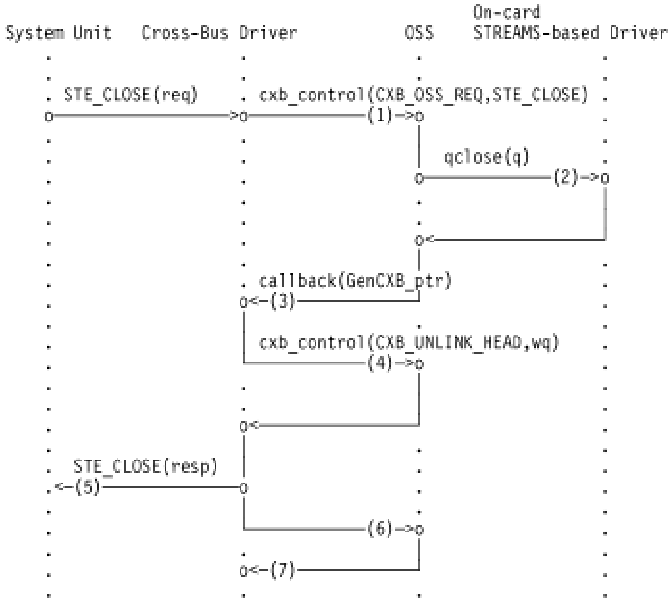

**Figure 5-4. Unlinking an On-card Stream Segment from the Cross-bus Driver**

(1) The system unit sends down a STE_CLOSE request formatted in an **ipcb** block. The cross-bus driver performs a CXB_OSS_REQ operation, using the cross-bus driver handle acquired as described in the registration sequence in Registering a Cross-Bus Driver on page 33.

(2) The OSS processes the close request. It internally marks the link between the stream head queues and the cross-bus driver's queues as unusable and calls the target driver's `queue close` routine.

(3) The OSS initiates a reply to the previous STE_CLOSE request through the cross-bus driver's callback routine. The memo field saved by the cross-bus driver from the STE_OPEN request is returned during the reply. The cross-bus driver recognizes that the incoming response is a STE_CLOSE.

(4) The cross-bus driver releases the memory attached to the q_ptr field of its write queue and unlinks its own queues by using the CXB_UNLINK_HEAD operation. Its pair of queues is freed at that time.

(5) The cross-bus driver responds to the STE_CLOSE request.

(6) The callback routine is returned.

(7) The first cxb_control function is returned.

# C Language Support

The operations, functions, and macros listed in the tables below provide information on how to develop your own cross-bus driver to interface between the system unit and the OSS. They are part of the OSS C language support and are described in this section. They follow the same rules as Standard Kernel Functions (SKFs) and OSS Kernel Functions (OKFs) explained in SKF Functions on page 17 and OSS Kernel Functions on page 20..

A cross-bus driver is not considered a STREAMS-based module/driver. Rather, it is considered a hybrid using a set of cross-bus driver commands and STREAMS service processing time for its cross-bus driver queues. Although functions described in the C language support for a STREAMS-based module/driver can be used by a cross-bus driver, the cxb_canputnext and cxb_putnext functions must be used in place of the canput and putnext functions for downstream communication from the cross-bus driver to the on-card stream. See Chapter 4, STREAMS-based Module/Driver Information for a list of other functions available to the cross-bus driver.

The Macro column specifies whether it is provided as an inline macro.

The Interrupt Handler column specifies whether it can be called from an interrupt handler.

| | | **Interrupt** | | **See** |
|---|---|---|---|---|
| **Operation** | **Macro** | **Handler** | **Description** | **Page** |
| Through the Cross-Bus Control Interface (cxb_control Function) | | | | |
| CXB_CBM_REQ | No | No | Sends a control request to the CBMS | 40 |
| CXB_LINK_HEAD | No | No | Links cross-bus driver queues to an on-card stream segment | 40 |
| CXB_OSS_REQ | No | No | Sends a control request to the OSS | 41 |
| CXB_REG_DRIVER | No | No | Dynamically registers a cross-bus driver into OSS. | 42 |
| CXB_UNLINK_HEAD | No | No | Unlinks cross-bus driver queues from an on-card stream segment | 43 |

| | Interrupt | | See |
|---|---|---|---|
| Macro | Handler | Description | Page |
| _SIZEOF_IPCB_EXTRA | Yes | Computes the control and `data extra reply` fields in an ipcb structure | 44 |
| _GET_OPEN_SSD | Yes | Extracts the on-card stream descriptor from a STE_OPEN response control block | 45 |
| _IS_STE_OPEN | Yes | Checks whether the command is a STE_OPEN | 45 |
| _IS_STE_CLOSE | Yes | Checks whether the command is a STE_CLOSE | 45 |
| _IS_IPCB_ERROR | Yes | Checks whether the ipcb response control block has an error set | 45 |

| | | Interrupt | | See |
|---|---|---|---|---|
| Function | Macro | Handler | Description | Page |
| Cross-Bus Data Interface | | | | |
| cxb_canputnext | No | No | Tests for available space in the next driver's write queue | 46 |
| cxb_putnext | No | No | Passes a message to the next write queue | 47 |

# cxb_control

Controls the interface between the cross-bus driver and OSS/CMBS components.

### Functional Prototype

```
int     cxb_control      ( cxbreq_t             *req);

typedef struct cxbreq
{
   unsigned long     operation;
   unsigned long     cxbhandle;
   union = {
      cxbreg_t       reg;
      cxblink_t      link;
      cxbunlink_t    unlink;
      cxbossreq_t    oss_req;
      cxbcbmreq_t    cbm_req;
   { cmd;
} cxbreq_t;
```

### Parameters

*exbhandle*

Input/output

*Input* This is the cross-bus driver handle obtained from a previous successful CXB_REG_DRIVER operation. This handle must be provided to all other operations defined in the cxb_control function (except the CXB_REG_DRIVER operation).

*Output* This is the cross-bus driver handle returned during a successful CXB_REG_DRIVER operation.

*operation*

Input. The following are available operations:

CXB_LINK_HEAD

Allocates and links cross-bus driver queues to an on-card stream segment (see CXB_LINK_HEAD).

CXB_OSS_REQ

Sends a control request to the OSS component (see CXB_OSS_REQ).

CXB_REG_DRIVER

Installs a cross-bus driver in the cross-bus switch table (see CXB_REG_DRIVER).

CXB_UNLINK_HEAD

Unlinks and deallocates cross-bus driver queues from an On-card Stream (see CXB_UNLINK_HEAD).

*returned value*

Output. On successful handling of the request, a value of zero is returned. An error code other than zero indicates the error.

### Error Codes

On failure, the function returns one of the following error codes:

| | |
|---|---|
| ENXIO | The cross-bus driver’s handle is invalid. |
| EINVAL | A parameter contains an unacceptable value. |
| EPERM | The operation is not permitted at this time. |

### Remarks

All operations must be sent to OSS from a service procedure during OSS processing time.

# CXB_LINK_HEAD

Allocates and links cross-bus driver queues to the on-card stream segment’s head queues.

### Functional Prototype

```
typedef struct cxblink
{
   dev_t                  devprofile;
   unsigned long          ssd;
   queue_t               *wq;
   unsigned long          reserved;
} cxblink_t;
```

### Parameters

*devprofile*

Input. Cross-bus driver device profile number. This is the device number assigned during a successful s96_devinst function call issued by the cross-bus driver to install a profile for its own operational queues. Queues are allocated during the link phase according to specifications contained in this profile. It is possible for a cross-bus driver to have multiple outstanding profiles (as many times as s96_devinst is called) and choose the operational one for this particular stream connection at link time.

For more information on how to create device profiles, see Device Profiles on page 32.

*ssd* Input. On-card stream descriptor. This value should be extracted by the cross-bus driver from the STE_OPEN response control block if the On-card Stream has been opened successfully. This represents the on-card stream segment that the cross-bus driver wants to link its queue pair with..

Setting *ssd* to NULL causes the cross-bus driver queues to be allocated but not linked with any on-card stream. This feature can be used for “super queues” allocation.

*wq* Output. Pointer to the allocated cross-bus driver write queue. The read queue pointer is obtained using the OTHERQ stream macro.

*reserved* Input. Reserved field (must be 0).

### Error Codes

| | |
|---|---|
| EBADF | The on-card stream’s descriptor is not a valid open stream descriptor. |
| ENOMEM | Not enough RadiSys ARTIC960 adapter memory available. |
| EFAULT | The *wq* pointer is not a valid cross-bus driver write queue pointer. |
| ENOENT | Device profile not installed. |

### Remarks

Cross-bus queues, once allocated, are managed totally by the cross-bus driver. Their deallocation has to take place using the CXB_UNLINK_HEAD operation at the time the on-card stream segment is closed. The q_ptr field is free to store instance-dependant information.

## CXB_OSS_REQ

Sends a control request to the OSS.

### Functional Prototype

```
typedef struct cxbossreq
{
   unsigned long         elmblock_len;
   struct ipcb          *elmblock_ptr;
   unsigned long         reserved;
} cxbossreq_t;
```

### Parameters

*elmblock_len*
Input. Size (in bytes) of the element control block, including its (optional) data and control fields.

*elmblock_ptr*
Input. Pointer to the beginning of the element control block containing the request parameters, formatted in an ipcb control block. The pointer value is set to the address of the elmblock union into the GenCXB memory block. The cross-bus driver must use the malloc function to acquire the GenCXB memory block. GenCXB memory block's ownership is transferred from the cross-bus driver to the OSS after the cxb_control function is called. The cross-bus driver must not perform any more operations on that GenCXB memory block, nor reuse the memory block after it calls the cxb_control function.

*reserved* Input. Reserved field (must be 0).

### Error Codes

| | |
|---|---|
| EBADF | The On-card stream's descriptor is not a valid open stream descriptor. |

### Remarks

If an error code is returned, the elmblock_ptr block's ownership is retained by the caller. OSS does not free the GenCXB memory block if it returns an error code value from this operation. The cross-bus driver is expected to provide a response to the error request's originator because OSS will *not* deliver any further response for such request in error.

STREAMS data messages should be sent using the cxb_putnext function described on page 47.

The CXB_OSS_REQ operation should be used only for mandatory control requests (see page 41). Optional requests can be formatted as STREAMS messages and sent using either the cxb_putnext (preferred method) or the cxb_control functions.

The following requests have to be carried using the CXB_OSS_REQ operation:

**Table 5-1. Requests for CXB_OSS_REQ Operation (Mandatory)**

| ConnectorPage | Page |
|---|---|
| STE_OPEN | 48 |
| STE_CLOSE | 49 |
| STE_XPUSH | 53 |
| STE_XPOP | 54 |
| STE_XLINK (Link) | 55 |
| STE_XLINK (Permanent Link) | 56 |
| STE_XUNLINK (Unlink) | 57 |
| STE_XUNLINK (Permanent Unlink) | 58 |
| STE_XLOOK | 59 |
| STE_XFIND | 60 |
| STE_XLIST | 61 |
| STE_XSETCLTIME | 62 |
| STE_XGETCLTIME | 63 |

The following requests can either be carried using the CXB_OSS_REQ function or directly as stream messages through the cxb_putnext function.

**Table 5-2. Requests for CXB_OSS_REQ Operation (Optional)**

| ConnectorPage | Page |
|---|---|
| M_FLUSH | 50 |
| M_READ | 51 |
| M_START | 51 |
| M_STOP | 52 |
| M_STARTI | 52 |
| M_STOPI | 53 |

# CXB_REG_DRIVER

## Description

Registers a callback routine into the OSS’s cross-bus driver table and stores the cross-bus driver’s memo value, which will be provided during the callback routine invocation. A cross-bus driver can register as many callback routines as it needs to accommodate its own design. The cxbreq_t structure field cxbhandle is initialized with the newly allocated driver’s handle if the operation is successful. This handle must be used for all other operations defined by the cxb_control function.

## Functional Prototype

```
typedef struct cxbreg
{
   void                (*cxbcallback_rtn)();
   unsigned long         cxbcallback_prm;
   unsigned long         reserved;
} cxbreg_t;
```

### Parameters

*cxbcallback_rtn*
Input. Specifies the pointer to the callback routine used by the OSS to deliver control messages to the cross-bus driver. The callback function's prototype is defined on page 31.

*cxbcallback_prm*
Input. Callback function parameter. This value is returned unaltered when the callback routine is invoked. The content of this variable is defined by how it is used. It is primarily intended as a pointer or index to aid the cross-bus driver in locating instance-specific information. Its use is optional.

*reserved* Input. Reserved field (must be 0).

### Error Codes

| | |
|---|---|
| ENOMEM | Not enough RadiSys ARTIC960 adapter memory available. |
| EMFILE | Too many cross-bus drivers registered. The maximum has been reached. |

### Remarks

If the operation is successful, the cxbhandle that is returned will be used in all other operations to correlate which cross-bus driver initiated the request. All responses and unsolicited indications will use this handle to retrieve the cross-bus driver's callback routine and parameters.

## CXB_UNLINK_HEAD

Unlinks the cross-bus driver queues from the on-card stream segment's head queues and deallocates them.

### Functional Prototype

```
typedef struct cxbunlink
{
   queue_t              *wq;
   unsigned long         reserved;
} cxbunlink_t;
```

### Parameters

*wq* Input. Pointer to the allocated cross-bus driver write queue. This is the queue pointer returned during a successful CXB_LINK_HEAD operation.

*reserved* Input. Reserved field (must be 0).

### Error Codes

| | |
|---|---|
| EFAULT | The *wq* pointer is not a valid cross-bus driver write queue pointer. |

### Remarks

The cross-bus driver should release any memory attached using the private q_ptr field prior to issuing this operation because queues' memory is freed by this operation.

# Macros

The following macros are defined to ease the implementation of cross-bus drivers when manipulating data structures of element control blocks.

| Macro | Page |
|---|---|
| _SIZEOF_IPCB_EXTRA | 44 |
| _GET_OPEN_SSD | 45 |
| _IS_STE_OPEN | 45 |
| _IS_STE_CLOSE | 45 |
| _IS_IPCB_ERROR | 45 |

## _SIZEOF_IPCB_EXTRA

Returns the size of the extra bytes to allocate for the optional response's control and data areas.

To minimize cross-bus memory occupation, the cross-bus driver might not want to allocate the extra bytes present at the tail of the elmblock union if these bytes are only used for the response. In that case, this macro computes the amount of extra memory needed at the bottom of the GenCXB structure to fulfill the response's requirements. The resulting size should be added to the size of the GenCXB structure before dynamically allocating the memory block.

### Functional Prototype

```
unsigned long   _SIZEOF_IPCB_EXTRA ( struct ipcb    *ipcbp);
```

### Parameters

*ipcbp* Input. Pointer to the beginning of the ipcb request block.

### Returns

Amount of bytes to append to the ipcb request block.

## _GET_OPEN_SSD

Extracts the descriptor value of the on-card stream's segment from a STE_OPEN's ipcb response block.

### Functional Prototype

```
unsigned long   _GET_OPEN_SSD ( struct ipcb    *ipcbp);
```

### Parameters

*ipcbp* Input. Pointer to the beginning of the ipcb response block.

### Returns

Descriptor value of on-card stream's segment.

## _IS_STE_OPEN

Checks if the ipcb control block carries a STE_OPEN command.

### Functional Prototype

```
unsigned long   _IS_STE_OPEN ( struct ipcb    *ipcbp);
```

### Parameters

*ipcbp* Input. Pointer to the beginning of the ipcb request/response block.

### Returns

The value 1 is returned if the control block is a STE_OPEN; otherwise, 0 is returned.

## _IS_STE_CLOSE

Checks if the **ipcb** block carries a STE_CLOSE request.

### Functional Prototype

```
unsigned long   _IS_STE_CLOSE ( struct ipcb    *ipcbp);
```

### Parameters

*ipcbp* Input. Pointer to the beginning of the ipcb request/response block.

### Returns

The value 1 is returned if the control block is a STE_CLOSE; otherwise, 0 is returned.

## _IS_IPCB_ERROR

Checks if the ipcb control block is in error.

### Description

For ipcb blocks carrying an error code, this macro retrieves the value from the *ipcb* response block.

### Functional Prototype

```
unsigned long   _IS_IPCB_ERROR ( struct ipcb    *ipcbp);
```

### Parameters

*ipcbp* Input. Pointer to the beginning of the ipcb element control block.

### Returns

The error number is returned; otherwise, 0 is returned.

## cxb_canputnext

Tests the next driver's write queue for availability

Returns the status of the next lower driver's write queue in the on-card stream's segment by checking its amount of bytes accumulated toward its configured high- and low-water marks. If the queue is not flow controlled, the cross-bus driver uses the cxb_putnext function to send data messages to the lower STREAMS-based module driver. High-priority messages should not be subject to flow control.

### Functional Prototype

```
int    cxb_canputnext   (queue_t              *wq);
```

### Parameters

*wq* Input. Pointer to the cross-bus driver allocated write queue. This queue has been previously allocated by a successful CXB_LINK_HEAD operation.

*returned value*

Output.

*1* This driver can accept data messages if the next driver's write queue is not flow controlled.

*0* The on-card stream's segment is temporarily flow controlled. When this condition occurs the cross-bus driver's write queue is automatically marked as *blocked*, and thus it will be back-enabled as soon as the target lower driver queue's byte count drops below its low water mark.

*–1* The on-card stream's segment is not able to process any data. There is a severe permanent condition needing control commands to clear.

### Remarks

Although canput appears to perform the same function, it should not be used to check a downstream queue. The cxb_canputnext function must be used instead.

# cxb_putnext

Calls the put procedure associated with the next STREAMS-based module/driver below the stream head, passing a message block into it. The message has to be formatted strictly as a stream message with no restriction from the standard UNIX SVR3/4 message format.

### Functional Prototype

```
int    cxb_putnext   (queue_t              *wq,
                      mblk_t               *mp);
```

### Parameters

*wq* Input. Pointer to the cross-bus driver-allocated write queue. This queue has been previously allocated by a successful CXB_LINK_HEAD operation.

*mp* Input. Stream message pointer. On successful completion, ownership of the message is transferred to the called STREAMS-based module/driver. The cross-bus driver should not attempt to reuse the same message pointer after it has successfully called the cxb_putnext function. (See returned value for exceptions.)

*returned value*

Output.

*1* Indicates that the message has been delivered.

*–1* Indicates that the on-card stream’s segment is not able to process any data. There is a severe permanent condition needing control commands to clear. In that case, and only in that case, ownership of the message pointer remains to the cross-bus driver after the call.

### Remarks

Although putnext appears to perform the same function, it should not be used to send data to a downstream queue. The cxb_putnext function must be used instead.

# Element Control Blocks Format

The cross-bus driver communicates with the On-card STREAMS Subsystem using element control blocks, either formatted as an ipcb block for OSS requests/responses, or as a cbms_cb block for CBMS requests/responses..

The current version of IBM’s OSS does not support cross-bus driver CBMS elements.

When no value is specified for a field, the field is not significant and does not need to be initialized (although setting a value of 0 for these is a recommended coding practice). A value specified in a request block and not used for the response is not altered.

# ipcb Blocks

Each ipcb control block has optional control and data parts at the tail of the structure. The size of this area can be determined using the macro _SIZEOF_IPCB_EXTRA.

## STE_OPEN — Open Stream

**Table 5-3. STE_OPEN**

| Field | Request | Response |
|---|---|---|
| mtype | STE_OPEN | |
| input | 1 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Device driver memo | |
| ctltype | cross-bus driver memo | Descriptor of opened stream |
| ctllen | Device name length (with ASCII termination) | 0 |
| datalen | 0 | |
| Flags | Stream’s flags | |
| extl | 0 | |
| ext2 | devno (minor) | devno (major/minor) |
| Control and Data parts | | |
| control part | Device name (with ASCII termination) | |
| data part | | |

The devno is updated in the response field with the card major number of the device name specified.

The cross-bus driver has to intercept the STE_OPEN if it needs to set a memo value associated with the opened stream. This memo value is returned unaltered to the cross-bus driver when its callback routine is invoked for this particular stream.

The device name to set in the request’s control section must be the name specified by the STREAMS-based driver during its installation using the s96_devinst function (*name* of the configuration structure).

## STE_CLOSE — Close Stream

**Table 5-4. STE_CLOSE**

| Field | Request | Response |
|---|---|---|
| mtype | STE_CLOSE | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

## STE_XSEND — Send Data

This is an immediate command. No further response is generated after the request unless an error occurs during processing.

The OSS offers the possibility to convey some sensitive high-priority messages into an **ipcb** block, thus ensuring more reliable delivery across the bus. The cross-bus driver can choose not to use the feature and still carry these high priority messages like a regular send. Note that the **datalen** field plays an important role in determining if a stream data block pointer is present or not.

**Table 5-5. STE_XSEND Data—Messages**

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | db_type field value from system unit's dblk_t block. |
| ctllen | 0 |
| datalen | 0 |
| flags | |
| ext1 | Stream data block pointer |
| ext2 | |

| | |
|---|---|
| Control and Data parts | |
| control part | |
| data part | |

The stream data block pointer is a flat RadiSys ARTIC960 adapter pointer value. Memory for data blocks and data buffers should have been acquired through CBMS and all links made with flat RadiSys ARTIC960 adapter pointer values.

1. It is unnecessary to transport the message block between the system unit and the card. It will be rebuilt by the OSS before the message is written downstream at the stream head, and by the SAL before the message is delivered through the STE_XRECEIVE response code.
2. Putting the db_type value in ctltype enables priority queueing in OSS's MPL.

### M_FLUSH

**Table 5-6. M_FLUSH—Message**

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 1 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_FLUSH |
| ctllen | 1 |
| datalen | –1 |
| flags | |
| ext1 | |
| ext2 | |
| Control and Data parts | |
| control part | Flush mode (Read,Write,Read/Write) |
| data part | |

### M_READ

**Table 5-7. M_READ—Message**

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_READ |
| ctllen | 0 |
| datalen | –1 |
| flags | |
| ext1 | Number of bytes to be read |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

### M_START

**Table 5-8. M_START—Message**

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_START |
| ctllen | 0 |
| datalen | –1 |
| flags | |
| ext1 | |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

## M_STOP

Table 5-9. M_STOP—Message

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_STOP |
| ctllen | 0 |
| datalen | –1 |
| flags | |
| ext1 | |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

## M_STARTI

Table 5-10. M_STARTI—Message

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_STARTI |
| ctllen | 0 |
| datalen | –1 |
| flags | |
| ext1 | |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

### M_STOPI

**Table 5-11. M_STOPI—Message**

| Field | Request |
|---|---|
| mtype | STE_XSEND |
| input | 0 |
| mseq | |
| error | |
| slotid | Stream descriptor |
| ctltype | M_STOPI |
| ctllen | 0 |
| datalen | –1 |
| flags | |
| ext1 | |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

## STE_XPUSH ioctl — Push Module

**Table 5-12. STE_XPUSH — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XPUSH | |
| input | 1 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | |
| ctllen | 0 | |
| datalen | Module name length (with ASCII termination) | 0 |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | Module name (with ASCII termination) | |

## STE_XPOP ioctl — Pop Module

Table 5-13. STE_XPOP — ioctl

| Field | Request | Response |
|---|---|---|
| mtype | STE_XPOP | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

## STE_XLINK ioctl — Link Driver

**Table 5-14. STE_XLINK — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XLINK | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Multiplexing driver’s Stream Descriptor | Device driver memo for mux stream |
| ctltype | Stream descriptor to connect below the multiplexor | Multiplexor ID number |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | 0 | |
| ext2 | Multiplexor ID number(l_index) | |
| Control and Data parts | | |
| control part | | |
| data part | | |

**Notes**

- If the request’s *ext1* parameter value is zero, the ioctl is a LINK ioctl. If it is 1, it is a PLINK ioctl.
- The request’s *`ext2`* parameter value is used when the system unit needs to assign the same link ID as the one it got from its own I_LINK system unit request. This feature is used mostly by system units supporting streams and is particularly useful so the application device driver does not have to keep a correspondence between system unit and card link IDs.

# iSTE_XLINK octl — Permanent Link Driver

**Table 5-15. STE_XLINK — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XLINK | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Multiplexing driver's stream descriptor | Device driver memo for mux stream |
| ctltype | Stream descriptor to connect below the multiplexor | Multiplexor ID number |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | 1 | |
| ext2 | Multiplexor ID number(l_index) | |
| Control and Data parts | | |
| control part | | |
| data part | | |

**Notes**

- If the request's *ext1* parameter value is zero, the ioctl is a LINK ioctl. If it is 1, it is a PLINK ioctl.
- The request's *ext2* parameter value is used when the system unit needs to assign the same link ID as the one it got from its own I_PLINK system unit request. This feature is mostly used by system units supporting streams and is particularly useful so the application device driver does not have to keep a correspondence between system unit and card link IDs.

## STE_XUNLINK ioctl — Unlink Driver

**Table 5-16. STE_XUNLINK — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XUNLINK | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Multiplexing driver’s stream descriptor | Device driver memo for mux stream |
| ctltype | Multiplexor ID number (or –1) | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | 0 | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

If the request’s *ext1* parameter value is zero, the ioctl is a LINK ioctl. If it is 1, it is a PLINK ioctl.

# STE_XUNLINK ioctl — Permanent Unlink Driver

**Table 5-17. STE_XUNLINK — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XUNLINK | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Multiplexing driver’s Stream Descriptor | Device driver memo for mux stream |
| ctltype | Multiplexor ID number (or –1) | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | 1 | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

If the request’s *ext1* parameter value is zero, the ioctl is a LINK ioctl. If it is 1, it is a PLINK ioctl.

## STE_XLOOK ioctl — Retrieve Top Module Name

**Table 5-18. STE_XLOOK — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XLOOK | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | |
| ctllen | 0 | |
| datalen | FMNAMESZ + 1 | Module name length (with ASCII termination) |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | Module name (with ASCII termination) |

## STE_XFIND ioctl — Find Module Name

**Table 5-19. STE_XFIND — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XFIND | |
| input | 1 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | 1 if present; otherwise, 0. |
| ctllen | 0 | |
| datalen | Module name length (with ASCII termination) | 0 |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | Module name (with ASCII termination) | |

## STE_XLIST ioctl — List Module Names

**Table 5-20. STE_XLIST — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XLIST | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | Number of entries to list | Number of entries listed |
| ctllen | 0 | |
| datalen | Maximum length of data part | |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | Modules list (**str_mlist** structures) |

In order to contain all modules names, the maximum length of *data part* specified in the request should normally be equal to:

```
Number of entries to list * sizeof(str_mlist)
```

## STE_XSETCLTIME ioctl — Set Close Time

**Table 5-21. STE_XSETCLTIME — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XSETCLTIME | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | Time delay value (in milliseconds) | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | | |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

## STE_XGETCLTIME ioctl — Get Close Time

**Table 5-22. STE_XGETCLTIME — ioctl**

| Field | Request | Response |
|---|---|---|
| mtype | STE_XGETCLTIME | |
| input | 0 | |
| mseq | | |
| error | | Error number or 0 |
| slotid | Stream descriptor | Device driver memo |
| ctltype | | |
| ctllen | 0 | |
| datalen | 0 | |
| flags | | |
| ext1 | | Time delay value (in milliseconds) |
| ext2 | | |
| Control and Data parts | | |
| control part | | |
| data part | | |

## STE_XRECEIVE Response Code — Receive Messages

**Table 5-23. STE_XRECEIVE — Response Code**

| Field | Response |
|---|---|
| mtype | STE_XRECEIVE |
| input | |
| mseq | |
| error | 0 |
| slotid | Device driver memo |
| ctltype | |
| ctllen | 0 |
| datalen | 0 |
| flags | |
| ext1 | Stream message pointer |
| ext2 | |
| Control and Data parts | |
| control part | |
| data part | |

The stream message pointer is a flat RadiSys ARTIC960 pointer value. Memory for the message and data blocks has been acquired through CBMS and should be deallocated using CBMS services. All internal links are made with flat RadiSys ARTIC960 pointer values.

# STREAMS Access Library 6

The STREAMS Access Library (SAL) is a tool set enabling a system unit kernel-mode driver to communicate with one (or many) RadiSys ARTIC960 adapters' On-card STREAMS Subsystems (OSSs). In this chapter, a kernel-mode driver is referred to as the *application device driver* (ADD). RadiSys ARTIC960 Support for AIX also provides the STREAMS960 Application Device Driver (called *S960ADD*), but SAL lets users communicate STREAMS messages to the adapter while communicating to non-STREAMS applications above from the AIX kernel space.

The provided API is designed to enable:

- Bridging a native UNIX SVR3/4 Stream environment from the system unit (for example, AIX) with the RadiSys ARTIC960 STREAMS located in the RadiSys ARTIC960 adapter. This is also called a *transparent service* (XPAR).
- Bridging a system unit not supporting the UNIX SVR3/4 Stream environment (for example, OS/2) with the RadiSys ARTIC960 STREAMS located in the RadiSys ARTIC960. This is also called a *non-transparent service* (NON-XPAR).

SALSER describes two kinds of services that are both included as part of the SAL system unit support:

- Stream services
- Memory services

The SAL underlying protocol to communicate with the adapter is not addressed in this book.

## C Language Support

The SAL C language support requires the operating system's standard libraries for device driver support. The SAL C language support consists of a header file, ***saluser.h***, providing prototypes to access the library routines and declares, and a definition of SAL command codes and structures. The ADD should include this header file whenever it calls a stream or memory service.

SAL commands available to ADDs are all prefixed with *s96_*.

## AIX Considerations

The SAL C language support provided for IBM AIX includes a library, **libsal.a**, containing all SAL routines. This library should be linked with other ADD object modules, along with the standard libraries of IBM AIX device drivers.

## OS/2 Considerations

The SAL C language support provided for IBM OS/2 includes a library, **sal.lib**, containing all SAL routines. This library should be statically linked with other ADD object modules along with the standard libraries of IBM OS/2 device drivers.

## Windows NT Considerations

The following header files are needed by the Windows NT driver. They must be included to provide the environment to access SAL functions by the Windows NT driver.

| | |
|---|---|
| ric.h | RadiSys ARTIC960 card-specific defines |
| salntstr.h | STREAMS defines for the Windows NT environment |
| oerrno.h | STREAMS error codes |
| saluser.h | Common functions across the various platforms that are provided by the SAL |
| salntusr.h | Functions specifically provided for the Windows NT environment |
| saldefs.h | Common SAL defines across the various platforms |
| salnt.h | Defines specific to the Windows NT SAL |
| cbmuser.h | Defines for card buffer management functions |

# Runtime Variables

The prototype definition of the following variables is in the **saluser.h** user include file. All these variables have default values which you can change.

**Table 6-1. SAL Runtime Variables**

| Variable | Description |
|---|---|
| sal_pipe_timeout | The SAL is monitoring the SCB (ric_scb.rel) pipe status and will timeout if it cannot enqueue any SCB control element to be processed by the RadiSys ARTIC960 adapter for more than a defined period of time. By default, the timer value is set to 5 seconds. It can be changed by the ADD using the externalized sal_pipe_timeout variable. (See **saluser.h** include file.) If timeout occurs, the SAL issues an STE_CONNECT response to the ADD, indicating that communication with the card is broken. |

| | |
|---|---|
| sal_deq_option | The SAL has two modes of operation: *interrupt dequeueing* and *service dequeueing*.<br>interrupt dequeueing<br>In this mode, the ADD response handler is called at *interrupt time*, running under the RadiSys ARTIC960 adapter device driver's off-level interrupt handler's time. Using this mode implies that the ADD's response handler and routines must be permanently resident in kernel space memory in order to function properly. Set the variable to SAL_INTERRUPT_DEQ to select this mode.<br>service dequeueing<br>In this mode, the application device driver's response handler is called at *STREAMS service queue time*, running under the operating system's streams scheduler's time. This is the default mode of operation and the associated variable is SAL_SERVICE_DEQUE. |
| sal_maxupstr_len | The SAL has a default message length for messages flowing upstream, set into the sal_maxupstr_len variable. The default length is SAL_MAX_REC_BUFF. (See the **saluser.h** include file for values.) The ADD can select a smaller or larger value by changing the value of the sal_maxupstr_len variable. Because downstream message length is dictated by the size of the buffer pool registered, it is not necessary to have a similar parameter for the downstream flow. |
| sal_ent_name | The SAL has a default SCB entity name when it registers its SCB entity. Providing an alternate SCB entity name enables the ADD to have two (or more) independent SCB communication pipes established between the SAL and the RadiSys ARTIC960 adapter device driver. The maximum length for the SCB entity name is fixed to MAX_RES_USER (ASCII string). |
| sal_cardmask | The SAL has a default cardmask of 0xFFFFFFFF. Each bit in this 32-bit unsigned long represents the status of cards numbered 0-31 in the system. A value of 1 indicates a disabled STREAMS environment on that card. You can use `smit` to change the value of this variable. |

## AIX Configuration

Configuration is the process of establishing communication between the ADD and the RadiSys ARTIC960 adapter device driver. The SAL is the component being directly attached to the RadiSys ARTIC960 adapter device driver. Once this communication is established successfully, SAL commands can be used and responses from the RadiSys ARTIC960 adapter's device driver will be received by the ADD.

The SAL is a library statically linked with the ADD. Through this library, the ADD has access to services described in *SAL Functions* on page 78. The SAL also has an interface with the operating system's streams subsystem after it is installed as a STREAMS-based driver.

AIX can use a daemon to hold the controlling stream open.

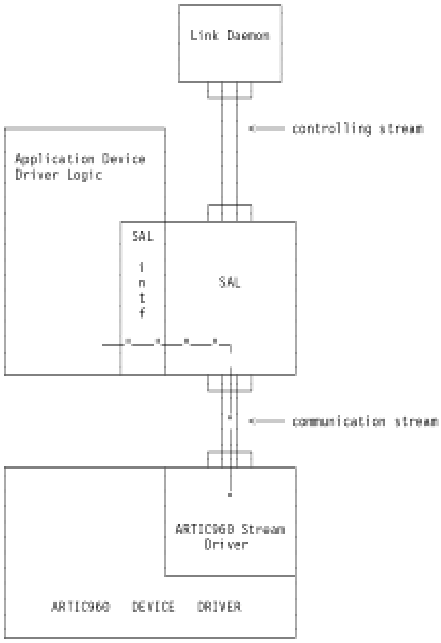

**Figure 6-1. AIX Application Device Driver Communication with RadiSys ARTIC960 Adapter Application Device Driver**

# OS/2 Configuration

There is no native streams support for OS/2. The configuration for the ADD would be specific to the environment within which the ADD operates. The configuration could be read from configuration files and/or be set by way of specific commands issued to the ADD.

In the specific case of an OS/2 implementation, the Media Access Control Driver extracts the port configuration from the file PROTOCOL.INI. This is typically done at boot time. The file is updated within the Multiple Protocol Transport Services (MPTS) environment. The Protocol Driver also binds, using the NDIS interface, with the MAC Driver at boot time. The MAC driver talks with the Protocol Driver by way of the NDIS/ANDIS interface on its upper layer and talks with the card components, on its lower layer, using the IDC (OS/2 Inter Driver Communication support) interface. The OS/2 SAL encapsulates the streams interface within its APIs.

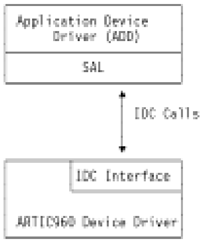

Figure 6-2. OS/2 ADD Communication with RadiSys ARTIC960 Device Driver

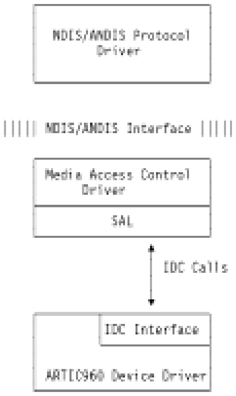
NDIS/ANDIS Protocol Driver
||||| NDIS/ANDIS Interface |||||
Media Access Control Driver
SAL
IDC Calls
IDC Interface
ARTIC960 Device Driver

Figure 6-3. OS/2 ADD (Media Access Control Driver) (Example)

## Installation of AIX SAL as a STREAMS-based Driver

To install the SAL as a STREAMS-based driver within the operating system streams subsystem, the ADD must explicitly call the stream installation (str_install) method provided by the operating system. The installation process registers the SAL’s streamtab structure within the operating system’s device table in order for user-level processes to open a stream with the SAL. The ADD has access to the SAL’s streamtab using the externalized name salmuxinfo (see **saluser.h** include file). Other installation parameters, such as the major node number, are directly provided to the ADD’s configuration routine.

## Linking the AIX SAL and the ARTIC960 Adapter Stream Driver

The SAL communicates with the RadiSys ARTIC960 adapter stream driver using a stream. To establish this stream, do the following:

1. Open a clone stream with the RadiSys ARTIC960 adapter device driver.
    - The RadiSys ARTIC960 adapter AIX Device Driver installs itself in the AIX Streams Subsystem using the name se960dd. The application is responsible for creating the device resource name corresponding to the major number assigned to this driver by the operating system (usually in the UNIX dev directory).
2. Open a clone or specific stream with the SAL stream driver. The ADD is responsible for giving a name to the stream extension installed and thus has total control over retrieving the major number allocated by the system.
3. Link the two streams together, the RadiSys ARTIC960 adapter stream driver being linked below the SAL. Then a link daemon must keep the link established. After the link is established, the link daemon cannot use system calls to communicate with the SAL through the controlling stream. The controlling stream must not be used further.

## Windows NT Configuration

The configuration for the ADD would be specific to the environment within which the ADD operates. The configuration could be read from configuration files, the registry, and/or be set by way of specific commands issued to the ADD. It is recommended that you do not use native streams support for Windows NT.

# 7 STREAMS Access Library Functions

This chapter describes two kinds of functions that are both included as part of the STREAMS Access Library (SAL) system unit support:

- Stream Functions — Used to open, monitor, transfer data, and close a stream with the RadiSys ARTIC960 adapter.
- Memory Functions — Used to allocate and free memory for streams data transfer with the RadiSys ARTIC960 adapter. The same functions are used by the RadiSys ARTIC960 adapter OSS but are sheltered from the STREAMS-based module/driver through the On-card Standard Kernel Function's (SKF) API.

## Stream Functions

The following are the Stream functions:

| Call | Description |
|---|---|
| s96_canput | Queries if the on-card stream is available to receive non-high priority messages from upstream |
| s96_close | Closes an on-card stream |
| s96_commstate | Queries the status of communication with an adapter |
| s96_couldput | Informs the SAL about a non-high priority message transmission upstream (flow control) |
| s96_ioctl | Performs an ioctl to the on-card stream |
| s96_open | Opens an on-card stream |
| s96_send | Sends a stream message to the on-card stream |

Memory Functions are described beginning on page 92. Each command is described with its prototype and restrictions.

## Memory Functions

The following are the memory functions:

| Call | Description |
|---|---|
| s96_bufcall | Registers a callback routine called when enough memory is available in the pool |
| s96_deregister | Deregisters a memory pool |
| s96_expand | Expands the available amount of memory in a pool |
| s96_free | Frees a block of memory |
| s96_info | Retrieves information about a pool |
| s96_register | Registers a memory pool |
| s96_reorg | Reorganizes a memory pool |
| s96_unbufcall | Cancels a pending s96_bufcall request |

The application device driver (ADD) cannot allocate memory from a memory pool because the SAL handles the data transfer to/from the system unit and the RadiSys ARTIC960 adapter.

See the memory functions beginning on page Memory Functions on page 92 for a description of each command with its prototype and restrictions.

# Functions Synchronization

Responses can be either immediate or asynchronous.

### Immediate

The application synchronization is automatically realized because the application resumes execution following the call to the function subroutine only after the function has been *completely* processed. The final error code is returned when the application resumes execution. An immediate function does not sleep in the function subroutine.

### Asynchronous

The application code continues execution after the call, parallel with the function processing on the RadiSys ARTIC960 adapter. The process is notified when the function operation completes by a call to a response handler, defined by the application. The final error code is returned in the response handler.

The SAL is responsible for the delivery of these responses and calls the application's response handler with parameters. These parameters are sufficient to correlate the response with a previous outstanding request so that the application can mark the request as completed with the accurate completion code value and any needed arguments.

The user must provide the response handler routine to handle responses from the RadiSys ARTIC960 STREAMS. A response is typically an incoming stream message flowing upstream and reaching the SAL. See Response Handler on page 74 for information on the s96_resphandler function.

For all requests, memory for function parameters is available for reuse as soon as control is returned to the application code after the SAL call.

Table 7-1 lists all SAL functions.

Immediate requests do not have an acronym defined because there is no asynchronous response associated with them.

**Table 7-1. SAL Functions**

| Function | Response Code | Synchronization | Page |
|---|---|---|---|
| **Stream Functions** | | | |
| s96_canput | N/A | immediate | 79 |
| s96_close | STE_CLOSE | asynchronous | 80 |
| s96_commstate | N/A | immediate | 81 |
| s96_couldput | N/A | immediate | 82 |
| s96_ioctl | STE_PUSH | asynchronous | 84 |

| Function | Response Code | Synchronization | Page |
|---|---|---|---|
| s96_ioctl | STE_POP | asynchronous | 84 |
| s96_ioctl | STE_LINK | asynchronous | 84 |
| s96_ioctl | STE_UNLINK | asynchronous | 85 |
| s96_ioctl | STE_LOOK | asynchronous | 85 |
| s96_ioctl | STE_FIND | asynchronous | 86 |
| s96_ioctl | STE_LIST | asynchronous | 86 |
| s96_ioctl | STE_SETCLTIME | asynchronous | 86 |
| s96_ioctl | STE_GETCLTIME | asynchronous | 86 |
| s96_open | STE_OPEN | asynchronous | 87 |
| s96_send | STE_XSEND[1] | immediate | 89 |
| **Memory Functions** | | | |
| s96_bufcall | N/A | immediate | 92 |
| s96_deregister | STE_DEREG | asynchronous | 94 |
| s96_expand | STE_EXPAND | asynchronous | 96 |
| s96_free | N/A | immediate | 97 |
| s96_info | N/A | immediate | 98 |
| s96_register | STE_REGISTER | asynchronous | 100 |
| s96_reorg | STE_REORG | asynchronous | 101 |
| s96_unbufcall | N/A | immediate | 102 |

[1] See s96_send for details on situations where a response can be provided on error conditions.

**Table 7-2. SAL Responses Received by the Response Handler**

| Function | Response Code | Description | Page |
|---|---|---|---|
| N/A | STE_XRECEIVE | Stream message received | 103 |
| N/A | STE_STOPXMIT | Stop sending messages downstream | 103 |
| N/A | STE_STARTXMIT | Restart sending messages downstream | 103 |
| N/A | STE_CONNECT | Informs on the status of an adapter | 104 |

# Response Handler

The response handler must be provided by the ADD. The SAL defines the s96_resphandler function with the following prototype.

### Functional Prototype

```
void    s96_resphandler      (unsigned long          memo,
                              unsigned long          command,
                              unsigned long          errcode,
                              unsigned long          arg1,
                              unsigned long          arg2,
                              unsigned long          reserved);
```

### Parameters

*memo* Memo value related to the on-card stream. This is the memo value given during an s96_open function by the application device driver. The memo identifies the on-card stream from where the response originates.

*command* Function code. Each response recalls the function code (see Table 7-1) and supported responses are listed in Table 7-2.

*errcode* Error code number. Possible error codes are listed with each request and indication commands. For a response, a value of 0 is returned if the request has been successfully processed. Otherwise, the error number qualifies the error. For a response, a value of 0 is always returned in this parameter.

*arg1/arg2*

Response additional information. This information is needed by the device driver to analyze the response. The data type of `arg1`/`arg2` depends on the particular response command code value, but it is either an integer (int) or a pointer to a response-specific information block.

*reserved* Reserved use by provider. Value is always 0.

### Remarks

The response handler is executing under a critical section of code. The application's response handler must never sleep. As for other handlers, the receive handler routine should be kept as short as possible. Otherwise, it may decrease device driver and/or system performance. If an extensive processing of the response needs to be performed, it may be necessary to queue the response data and service it at a lower processing level.

All pointer parameters passed to the response handler are valid until the response handler is returned. A pointer parameter might be saved and reused after the response handler is returned, unless otherwise instructed in the routine description.

# Programming Notes

In order to separate SAL and ADD variables and functions, the SAL prefixes all its variables and function names with *sal_*. The ADD should follow a similar convention to avoid using the SAL prefix and thereby prevent conflicts with the SAL variables.

# Priority Messages

The following list of message types are high-priority messages in the RadiSys ARTIC960 STREAMS.

**Table 7-3. High-priority Messages**

| Primitive | Origin | Direction of Flow | Comments |
|---|---|---|---|
| M_COPYIN | MD | Upstream | Not supported. |
| M_COPYOUT | MD | Upstream | Not supported. |
| M_ERROR | MD | Upstream | |
| M_FLUSH | WSH / MD | Upstream / downstream | |
| M_HANGUP | MD | Upstream | |
| M_IOCACK | MD | Upstream | |
| M_IOCDATA | WSH | Downstream | Not supported. |
| M_IOCNAK | MD | Upstream | |
| M_PCPROTO | WSH / MD | Upstream / downstream | See note following this table for special considerations. |
| M_PCRSE | MD | Upstream / downstream | Message freed by SAL if sent downstream; by OSS, if sent upstream. |
| M_PCSIG | MD | Upstream | |
| M_READ | WSH | Downstream | |
| M_START / M_STOP | MD | Downstream | |
| M_STARTI / M_STOPI | MD | Downstream | |

**Note:** MD = module or driver; WSH = Write of the stream head.

**Special Considerations:** All supported high-priority messages types, except the M_PCPROTO, are transferred from the system unit to the RadiSys ARTIC960 STREAMS without using on-card memory. The SAL determines which message type is being sent and extracts the message's parameters to initialize an SCB's entity-to-entity field before sending the SCB element to the RadiSys ARTIC960 STREAMS.

The M_PCPROTO message type is treated differently because its potential length may not fit the SCB element's requirements. The SAL uses an internal high-priority on-card memory pool to transfer the M_PCPROTO message from the system unit to the RadiSys ARTIC960 STREAMS. Depending on the condition, the s96_send function returns the following error codes.

| | |
|---|---|
| AL_EAGAINS | In the event that the SCB element carrying the high-priority message cannot be transferred successfully to the RadiSys ARTIC960 adapter. |
| ENOMEM | If memory cannot be allocated from the high-priority on-card memory pool. |

If either condition occurs, the ADD must queue the high-priority message and retry it. Queueing the high-priority message in a stream function queue forces the queue to be serviced. Following this protocol avoids having to have an asynchronous indication trigger the s96_send function retry.

For transfers of high-priority messages from the RadiSys ARTIC960 STREAMS to the system unit (including M_PCPROTO), the SAL receives the stream message as it is originally built by the on-card STREAMS-based module/driver. The message is delivered to the ADD through the asynchronous response handler, using the STE_XRECEIVE response code, in the same manner as for other stream messages.

# Flow Control

Flow control applies to downstream and upstream message flows.

- The *downstream flow* is directed from the system unit to the RadiSys ARTIC960 adapter.
- The *upstream flow* is directed from the RadiSys ARTIC960 adapter to the system unit.

Flow control is based on a certain number of outstanding low-priority data messages being exchanged between the RadiSys ARTIC960 adapter and the system unit per stream opened, as well as for the SCB channel established between those two peers and used to carry information elements back and forth.

## Downstream Flow

The on-card stream receives data messages from the system unit's stream and uses the stream's flow control mechanism before forwarding messages to the next on-card STREAMS-based module/driver by invoking the *canput()* SKF API. When the next STREAMS-based module/driver reaches its high-water mark for its write service queue, the on-card stream head begins queueing received data messages and servicing them when it gets back-enabled after the next STREAMS-based module/driverwater mark goes below its low level. The system unit's SAL increments the outstanding number of data messages each time a successful s96_send function is performed by the ADD. This same number is decremented by the OSS when it can successfully forward the data message to the next STREAMS-based module/driver.

As OSS is queueing data messages when the next on-card STREAMS-based module/driver is flow controlled, the number of outstanding data messages reaches its own high-water mark. (See page 12 for information on the STRSCBQUEUED OSS parameter.) When this happens, the SAL calls the ADD's response handler with a STE_STOPXMIT response code for the stream being flow controlled.

The ADD should stop sending data messages downstream until it is instructed to do so by receiving a STE_STARTXMIT response code through its response handler for that stream. Failure of the ADD to comply with this protocol provokes depletion of other resources. In the event of an attempt to send other data messages downstream, and if the flow control condition persists, the SAL issues another STE_STOPXMIT response code for each attempt. However, the data message will be handled as long as other contributing resources are available.

The STE_STARTXMIT response code comes from the OSS when, while decrementing the outstanding number of data message, it finds that the count is now below its low-water mark and the SAL has issued a STE_STOPXMIT response code to the ADD.

### Global Communication Channel Flow

The Global Communication Channel is the SCB pipe established between the SAL and the OSS. This channel can also get congested during transfer peaks. This channel is not under control of the ADD, but its state will lead to control all data traffic back and forth between the RadiSys ARTIC960 adapter and the system unit for all streams.

- When the channel high-water mark is reached (see page 11 for information on the MAXSCBQUEUED OSS parameter), the ADD gets a STE_STOPXMIT response code through its response handler as it normally would for stream congestion, and reacts in the same manner as described in Downstream Flow on page 76.
- When the channel low-water mark is reached again, the SAL gives the ADD an event indicating the Global Communication Channel is available again. An STE_CONNECT response code, using subcode S96_COMM_XON, is delivered through the ADD response handler. The ADD should retry *all* streams that have been blocked through a STE_STOPXMIT response code because this S96_COMM_XON is a global event for all streams.

## Upstream Flow

The ADD receives data messages from the OSS through its response handler using the STE_XRECEIVE response code. The expected STE_XRECEIVE response code processing is to forward the data message to the next STREAMS-based module/driver or queue it for further processing. It is not possible for the ADD to refuse a data message delivered through its response handler. On return from the response handler the SAL considers the message is being handled by the ADD. The OSS increments the outstanding number of data messages each time it can successfully send a data message upstream. This same number is decremented by the ADD when it can successfully forward the data message to the next STREAMS-based module/driver upstream. As ADD is queueing data messages when the next STREAMS-based module/driver is flow controlled, the number of outstanding data messages reach its own high-water mark. (See page *12* for information on the STRSCBQUEUED parameter.) When this happens, the OSS begins queueing data messages coming from the STREAMS-based module/driver attached below its stream head, which in turn provokes this driver to be flow-controlled by the RadiSys ARTIC960 STREAMS flow control mechanism. This situation remains until the OSS receives a STE_XSTARTXMIT response code.

When the ADD starts emptying its data message queue to forward the message upstream it has to call the s96_couldput function. This function takes care of decrementing the number of outstanding data messages for the stream.

The STE_XSTARTXMIT response code comes from the SAL during the s96_couldput function processing (if, while) decrementing the outstanding number of data messages, it finds that the count is now below its low-water mark and the OSS is currently flow controlling the on-card stream.

# SAL Functions

The SAL's functions provide the capability for a system unit's device driver to open, maintain and close RadiSys ARTIC960 adapter on-card streams.

All function names are prefixed with *s96_*, which identifies functions directed to the RadiSys ARTIC960 STREAMS.

Unused bits in parameters should be cleared to 0 for future use.

The *Functional Prototype* section of each function description includes the type of each parameter. The following types are defined.

| | |
|---|---|
| *int* | Signed 32 bits integer |
| *dev_t* | Unsigned 32 bits integer |
| *mblk_t* | Stream message block structure |
| *dblk_t* | Stream data block structure |

The *Response Description* section of each function description has the response code that indicates which component is responsible for that function's initialization. The following naming convention is used.

| **Value** | **Parameter Loaded By** |
|---|---|
| input | Application process/device driver |
| output | SAL and/or OSS subroutines |

Any *output* response code parameter should be saved by the application process/device driver as it may be requested in other functions as *input*.

For *asynchronous functions*, the request's return is relevant to the SAL's ability to successfully handle and transmit the request to the RadiSys ARTIC960 adapter. Therefore the final return code, delivered when the request has been completely handled by the RadiSys ARTIC960 adapter, is known when the response handler is called.

If a non-zero error code is returned at the time of the request, it is assumed that the request has failed and that no further asynchronous response is delivered for this particular request.

# Stream Functions

The following sections describe each stream function with its prototype and restrictions.

## s96_canput

Queries if an on-card stream is available for messages to be received.

### Functional Prototype

```
unsigned long   s96_canput      (unsigned long          sd,
                                 unsigned long          reserved);
```

### Parameters

*sd* Input. On-card stream’s descriptor. This is the descriptor obtained from a successful s96_open function.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. If the stream is available, a value of 1 is returned. Otherwise, a value of 0 is returned.

### Remarks

The s96_canput function determines if the on-card stream is available to receive more non-high-priority messages.

## s96_close

Closes an on-card stream access if other accesses remain after this close.

The last close for the on-card stream causes the stream associated with *`sd`* to be dismantled. If there are data on the modules' write queue, the close operation waits up to 15 seconds per module/driver for any output to drain before dismantling the stream. The time delay can be changed using a STE_SETCLTIME s96_ioctl function.

### Functional Prototype

```
unsigned long   s96_close       (unsigned long          sd,
                                  unsigned long          reserved);
```

### Parameters

| | |
|---|---|
| *sd* | Input. On-card stream's descriptor. This is the descriptor obtained from a successful s96_open function. |
| *reserved* | Input. Reserved use by provider. Value must be 0. |

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The stream access associated with the descriptor is closed unless one or more of the following error codes are true.

| | |
|---|---|
| SAL_EBADF | The on-card stream's descriptor (*`sd`*) is not a valid open stream descriptor. |
| SAL_EINTR | A signal was caught during the close operation. |
| SAL_EAGAIN | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |

### Response Description

The *`command`* parameter is set to STE_CLOSE. Both arguments, *`arg1`* and *`arg2`*, are 0.

## s96_commstate

Queries or changes the status of communication with an RadiSys ARTIC960 STREAMS.

### Functional Prototype

```
unsigned long   s96_commstate   (int                      cardnum,
                                 unsigned long          *commstate);
```

### Parameters

*cardnum* Input. The logical RadiSys ARTIC960 adapter number. Valid adapter numbers range from 0–15.

*commstate*

Input/output. Pointer to the adapter communication status returned or new communication state to set. The following value can be used.

S96_COMM_QUERY

Input. Queries the status of the RadiSys ARTIC960 adapter number specified by cardnum regarding communication with the RadiSys ARTIC960 STREAMS. On return, the ***commstate*** parameter is being updated by the current status. See page 73 (STE_CONNECT) for details on S96_COMM_UP and S96_COMM_DOWN states. No asynchronous response is provided after the call is returned.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The function succeeds unless the following error code is true.

| | |
|---|---|
| SAL_EINVAL | Invalid parameter specified. |

### Remarks

None

## s96_couldput

Informs the SAL about a successful non-high-priority message transmission upstream.

### Functional Prototype

```
unsigned long   s96_couldput    (unsigned long          sd,
                                  unsigned long          reserved);
```

### Parameters

*sd* Input. On-card stream's descriptor. This is the descriptor obtained from a successful s96_open function.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. If the stream is available to receive messages, then on successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The function succeeds unless one or more of the following error codes are true:

| | |
|---|---|
| SAL_EINVAL | Invalid parameter specified. |
| SAL_EBADF | The on-card stream's descriptor (*sd*) is not a valid open stream descriptor. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_ENXIO | I/O error. |

### Remarks

The s96_couldput function must be called by the ADD when it can successfully forward a stream message received from its s96_resphandler function with the STE_XRECEIVE response code.

Only non-high-priority data messages are considered for flow control. Thus, the ADD should check the first data block's db_type field to make sure it is a low-priority data message (value lower than QPCTL). It should then call the s96_couldput function, giving the on-card stream descriptor (*sd*) for the appropriate stream.

Only non-high-priority data messages are considered by flow control. The s96_couldput function should be called only once for those non-high-priority data messages.

For maximum flow control efficiency, the ADD should call the s96_couldput function at the same time it passes the stream message to the next STREAMS-based module/driver upstream, especially if the ADD has some means of internally queueing stream messages.

## s96_ioctl

Performs an ioctl on an on-card stream.

### Functional Prototype

```
unsigned long   s96_ioctl       (unsigned long          sd,
                                 unsigned long          iocmd,
                                 unsigned long          arg,
                                 unsigned long          reserved);
```

### Parameters

*sd* Input. On-card stream's descriptor. This is the descriptor obtained from a successful s96_open function.

*iocmd* Input. Ioctl request value. Currently supported ioctls are :

| | |
|---|---|
| STE_PUSH | Pushes a module to the top of the on-card stream. See page 84 for more information. |
| STE_POP | Removes a module from the top of the on-card stream. See page 84 for more information. |
| STE_LINK | Links two on-card streams. See page 84 for more information. |
| STE_UNLINK | Unlinks two on-card streams. See page 84 for more information. |
| STE_LOOK | Retrieves the name of the topmost module present on the on-card stream. See page 84 for more information. |
| STE_FIND | Checks if a specific module name is present on the on-card stream. See page 84 for more information. |
| STE_LIST | Lists all module names on the on-card stream. See page 84 for more information. |
| STE_SETCLTIME | Sets the closing time delay allowing write queues to drain. See page 84 for more information. |
| STE_GETCLTIME | Returns the closing time delay. |

*arg* Input. Ioctl additional information. This information is needed by the device to perform the requested function. The data type of `arg` depends on the particular request value, but it is either an integer or a pointer to a request-specific information block.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The ioctl succeeds unless one or more of the following error codes are true or one or more of the request-specific error codes are true.

| | |
|---|---|
| SAL_EBADF | The on-card stream's descriptor is not a valid open stream descriptor. |
| SAL_EINTR | A signal was caught during the ioctl operation. |
| SAL_ENXIO | Hangup received from the on-card stream. |
| SAL_EAGAIN | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |

### Remarks

All incoming ioctls already formatted as messages (M_IOCTL), and not trapped by the ADD, can be forwarded to the on-card stream using the s96_send function. They are then transported transparently to the target on-card STREAMS-based module/driver without being interpreted by the RadiSys ARTIC960 STREAMS.

The following sections describe each supported ioctl. Request-specific error codes are valid for both the request and response phases.

**Table 7-4. Descriptions of Supported s96_ioctl Commands**

| Description | Response Description | Error Codes |
|---|---|---|
| **STE_PUSH ioctl** | | |
| Pushes a STREAMS-based module, whose name is pointed to by the *arg* parameter, onto the top of the on-card stream. It then calls the queue open routine of the newly-pushed module.<br>The maximum length for a module name (ASCII termination excluded) is set to FMNAMESZ. | The *command* parameter is set to STE_PUSH. Both arguments, *arg1* and *arg2*, are 0. | SAL_EINVAL<br>Incorrect module name.<br>SAL_EFAULT<br>The `arg` parameter points outside the allocated address space.<br>SAL_ENXIO<br>The open routine of the new module failed. |
| **STE_POP ioctl** | | |
| Removes a STREAMS-based module previously pushed from the top of the on-card stream. The value of the `arg` parameter should be 0. | The `command` parameter is set to STE_POP. Both arguments, *arg1* and *arg2*, are 0. | SAL_EINVAL<br>No module is present in the on-card stream. |
| **STE_LINK ioctl** | | |

| Description | Response Description | Error Codes |
|---|---|---|
| The _stelink structure contains the following parameters.<br>`unsigned long   l_sdbot;`<br>`int             l_index;`<br>l_sdbot<br>The on-card stream descriptor (*sd*) of the stream connected to another on-card STREAMS-based driver. The stream designated in this parameter gets connected below the multiplexing driver.<br>l_index<br>The link index to assign to this link. If a link index value of 0 is passed, a link index is assigned by the OSS and propagated to the on-card STREAMS multiplexer driver. | The *command* parameter is set to STE_LINK.<br>The *arg1* parameter contains the multiplexer ID number (an identifier used to disconnect the multiplexer). (See the following STE_UNLINK ioctl.) The *arg2* parameter is 0. | SAL_ETIME<br>Time-out before acknowledgment message received.<br>SAL_EAGAIN<br>Temporarily unable to allocate storage to perform the operation.<br>SAL_ENOSR<br>Unable to allocate storage to perform the operation because of insufficient OSS memory resources.<br>SAL_EBADF<br>The *arg* on-card stream's descriptor (*sd*) is not a valid open stream descriptor.<br>SAL_EINVAL<br>The specified link operation would cause a cycle in the resulting configuration. |
| **STE_UNLINK** | | |
| Unlinks two on-card streams, where *sd* is the on-card stream descriptor of the stream connected to the multiplexing driver, and the *arg* parameter is the multiplexer ID number that was returned by the STE_LINK operation. If the value of the *arg* parameter is –1, all on-card streams that were linked to the *sd* on-card stream are disconnected.<br>STE_UNLINK *must* be used to break links established using the STE_LINK ioctl. | The *command* parameter is set to STE_UNLINK. Both arguments, *arg1* and *arg2*, are 0. | SAL_ETIME<br>Time-out before acknowledgment message received.<br>SAL_ENOSR<br>Unable to allocate storage to perform the operation because of insufficient OSS memory resources.<br>SAL_EINVAL<br>One of the following:<br>• The *arg* parameter is an invalid multiplexer ID number<br>• The *sd* descriptor is not the on-card stream on which the STE_LINK operation that returned the *arg* value was performed. |
| **STE_LOOK** | | |

| Description | Response Description | Error Codes |
|---|---|---|
| Retrieves the name of the module located at the top of the on-card stream. The *arg* parameter should be set to 0. | The *command* parameter is set to STE_LOOK.<br>The *arg1* parameter is a pointer to a null-terminated string containing the module name retrieved. The *arg2* parameter is 0. | SAL_EINVAL<br>No module is present in the on-card stream. |
| **STE_FIND** | | |
| Checks if a specific module name is present on the on-card stream.<br>Checks the names of all modules currently present on the on-card stream against the name pointed to by the *arg* parameter. The name pointed to by *arg* should be an ASCII-terminated string. | The *command* parameter is set to STE_FIND.<br>Depending on whether the named module is present in the on-card stream, the *arg1* parameter is set as follows:<br>• 1—if present<br>• 0—if not present.<br>The *arg2* parameter is 0. | SAL_EFAULT<br>The *arg* parameter points outside the allocated address space.<br>SAL_EINVAL<br>The *arg* parameter does not contain a valid module name. |
| **STE_LIST** | | |
| Lists all the module names present in the on-card stream. If the value of the *arg* parameter is null, only the number of modules present in the on-card stream are returned. If the *arg* parameter contains the number of entries to list, the list of modules is returned. | The *command* parameter is set to STE_LIST.<br>If the request contained a null *arg*, the *arg1* parameter is a value indicating the number of modules present on the on-card stream and the *arg2* parameter is set to null. Otherwise, the *arg1* parameter indicates the number of modules listed and *arg2* points to an area containing *arg1* number of str_mlist structures contiguous in memory.<br>The str_mlist structure contains the following parameter.<br>`char`<br>`modname[FMNAMESZ+1];` | SAL_EAGAIN<br>Unable to allocate buffers. |
| **STE_SETCLTIME** | | |
| Sets the closing time delay allowing write queues to drain. Before closing each module and driver, the OSS delays closing for the specified length of time to allow the data to drain normally. Any data left after the delay is flushed.<br>The *arg* parameter contains the number of milliseconds to delay. The value is rounded up to the next multiple of 10 milliseconds. By default, the time delay is set to 15 seconds. | The *command* parameter is set to STE_SETCLTIME. Both arguments, *arg1* and *arg2*, are 0. | SAL_EINVAL<br>The time value in the *arg* parameter is invalid. |
| **STE_GETCLTIME** | | |

| Description | Response Description | Error Codes |
|---|---|---|
| Returns the closing time delay, in milliseconds, when an on-card stream is closing. | The *command* parameter is set to STE_GETCLTIME.<br>The *arg1* parameter contains the returned time delay in milliseconds. The *arg2* parameter is 0. | No specific error code defined. |

## s96_open

Opens an on-card stream access.

Opens a stream to a device (devname) located on the RadiSys ARTIC960 adapter number (*cardnum*). If a cloned stream is requested, the stream flag is set to CLONEOPEN. Otherwise, the minor portion of the device number (*devno*) specifies the specific device resource to open.

### Functional Prototype

```
unsigned long   s96_open        (int                    cardnum,
                                 char                  *devname,
                                 dev_t                 *devno,
                                 int                    sflag,
                                 unsigned long          memo,
                                 unsigned long          reserved);
```

### Parameters

*cardnum* Input. The logical RadiSys ARTIC960 adapter number where the device is defined. Valid adapter numbers range from 0–15.

*devname* Input. Pointer to the device name to open (ASCII string). This is a device name defined by the device resource configuration process, for whom the system unit device resource configuration process has assigned a major node number. The RadiSys ARTIC960 adapter also needs to have this specific device name configured in its device table. The function fails if the specified device name cannot be found in the target RadiSys ARTIC960 adapter’s number.

The maximum length for a device name (ASCII termination excluded) is set to FMNAMESZ (defined in the C language support **include** file).

*devno* Input. Pointer to the major/minor node number.

On input, only the minor portion of the device number is significant and holds the minor node number of the device resource to open. The minor number value is not significant if a CLONEOPEN is requested. (See the *sflag* parameter description.) The minor number, if not CLONEOPEN, is transparently forwarded by the RadiSys ARTIC960 STREAMS to the on-card STREAMS-based driver’s queue open routine during the driver open sequence.

Macros like makedev, major, and minor ease the manipulation and construction of the device number.

*sflag* Input. Stream’s flags. The following flags are supported.

0x00

Normal driver open. The minor node number specified in the *devno* field is significant.

0x2 (CLONEOPEN)

Clone driver open. The minor node number specified in the *devno* field is *not* significant on input. However, it is initialized before returning to the caller with the device resource’s card minor node number assigned by the On-card STREAMS-based driver.

*memo* Input. Correlation value. This value is provided during asynchronous notification of events. The content of this variable is implementation defined, but is primarily intended as a pointer or an index that would aid the device driver in locating instance-specific information.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The device resource specified is opened unless one or more of the following error codes is true.

| | |
|---|---|
| SAL_EINVAL | Invalid RadiSys ARTIC960 adapter number specified. |
| SAL_ENAMETOOLONG | The length of the device name argument exceeds (FMNAMESZ) bytes. |
| SAL_EAGAIN | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |
| SAL_EMFILE | The process has too many open files. |
| SAL_ENXIO | An on-card STREAMS-based module or driver open routine failed. |
| SAL_ENXIO | The specified stream resource is already opened. |
| SAL_ENOSR | Unable to allocate a stream. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_ENOMEM | Not enough RadiSys ARTIC960 adapter memory available. |
| SAL_ENOENT | Device name not found. |
| SAL_EBUSY | Device is in the closing state. |

### Remarks

The usual open flags O_NDELAY, O_NONBLOCK, O_RDONLY, O_WRONLY, O_RDWR are not applicable in the open operation. However, by default, the on-card stream is opened with O_NDELAY reset and O_RDWR set.

The effective user ID (uid) and group ID (gid) are both set to (0,0) when passed to the RadiSys ARTIC960 STREAMS driver’s queue open routine.

### Response Description

The *command* parameter is set to STE_OPEN.

On successful completion, the *arg1* parameter is the on-card stream's descriptor (*sd*) value for the opened on-card stream. This stream descriptor must be used whenever a subsequent command applies to this on-card stream.

The *arg2* parameter is a pointer to the adapter device number (major/minor node number) corresponding to the device resource opened. The device number is a dev_t structure, where the major portion holds the adapter major node number corresponding to the device name specified in the request, and the minor portion holds the card minor node number assigned by the on-card STREAMS-based driver if the request specified a clone open.

## s96_send

Sends a stream message to an on-card stream's segment.

Sends a message, pointed to by *mp*, to an on-card stream. The message must be formatted according to the system unit's operating system stream environment version. The application is responsible for freeing any of the memory pointed to by *mp*. On successful completion of the send request, the message is considered sent, giving the application the opportunity to free the memory associated with the message as soon as the function is returned..

This is an immediate function; no further response is generated after the call is returned.

### Functional Prototype

```
unsigned long   s96_send        (unsigned long          sd,
                                 unsigned long          ehandle,
                                 mblk_t                *mp,
                                 void                 **adp,
                                 unsigned long          reserved);
```

### Parameters

*sd* Input. On-card stream's descriptor. This is the descriptor obtained from a successful s96_open function.

*ehandle* Input. Entity's on-card memory pool handle. This is the handle obtained from a successful s96_register function.

*mp* Input. Pointer to the operating system's stream message block structure. This is a pointer to the stream message block structure (mblk_t), which contains a pointer to the data block structure (*dblk_t*).

*adp* Input/output. Address of an additional data parameter returned, depending on the return code value.

- When the SAL_EAGAIN return code is returned, on output, the *adp* pointer value is initialized with the address of the on-card message. The ADD should save the *adp* pointer value to provide it as input when the send operation is retried later. Using this feature optimizes performance by not reallocating on-card memory each time a send operation is retried because of a SCB-pipe-full condition. Instead, the on-card memory is held as long as the ADD does not perform a specific s96_free function on the on-card memory held, using this pointer value. (The value is a flat RadiSys ARTIC960 adapter address.)

- When the SAL_ENOMEM return code is returned, on output, the *adp* pointer value is initialized with the amount of data (in bytes) required to perform the on-card memory allocation. This value serves as input to the s96_bufcall function.
- On input, if the *adp* pointer value is not NULL, it is used as the on-card memory location where the data block and buffer reside. The value provided as input should be the one previously returned from a send operation regarding the same message. (See s96_free for more information.)

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The stream message is sent unless one or more of the following error codes are true.

| | |
|---|---|
| SAL_EBADF | The on-card stream's descriptor (*sd*) is not a valid open stream descriptor. |
| SAL_EINVAL | The entity's on-card memory pool handle is invalid. |
| SAL_EFAULT | Invalid message pointer (*mp*). |
| SAL_ENOMEM | Not enough memory in the entity's on-card memory pool specified. |
| SAL_EAGAIN | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_ENXIO | The total message's length exceeds the maximum on-card memory buffer capacity. (See s96_register for more information.) |

### Remarks

The mblk_t and dblk_t structures are UNIX SVR3/4 streams structures. However, each operating system stream environment has its own set of internal reserved fields in these structures. The SAL is built to correspond with the system unit's operating system type and version. Any change (upgrade) to the system unit's operating system type and/or version that would result in a change to the mblk_t or dblk_t structures format, may require a change (upgrade) to the SAL version handling these new formats.

The RadiSys ARTIC960 STREAMS does not support queue banding. Only normal- and high-priority messages are handled. If a message with a priority band greater than 0 is sent to the RadiSys ARTIC960 STREAMS, the message priority is forced to 0 (normal) before it is processed by the RadiSys ARTIC960 adapter STREAMS-based module/driver. Therefore, upstream messages originating from the RadiSys ARTIC960 STREAMS are either normal or high-priority messages only, and the db_band field value is set to 0 in both cases.

The ADD can alter the message priority before it forwards the message to the next upstream module/driver by changing the db_band field value if desired..

If the message type (db_type field) is an ioctl (M_IOCTL), the ioctl command (ioc_cmd field) values ranging from 0x5300 to 0x53FF are RadiSys ARTIC960 adapter reserved values that should not be used by the ADD logic.

### Response Description

The *command* parameter is set to STE_XSEND.

A response is generated *only* if the send operation fails in the RadiSys ARTIC960 adapter.

If the SAL_EBADF error code is returned, the *memo* is set to the invalid on-card stream's descriptor (*sd*) passed during the s96_send() function call. Both arguments (*arg1* and *arg2*) are 0..

ADF stands for Adapter Description File.

# Memory Functions

The following sections describe each memory function with its prototype and restrictions.

## s96_bufcall

Registers a function to be called when a certain amount of bytes is available in the on-card memory pool.

The function `cfunc` is registered and called back when at least `csize` bytes are available for allocation in the pool. The function is called with `cparm` as the argument.

### Functional Prototype

```
unsigned long   s96_bufcall   (unsigned long          handle,
                               unsigned long          csize,
                               b_callfunc             cfunc,
                               unsigned long          cparm,
                               unsigned long          reserved);
```

### Parameters

*handle* Input. On-card memory pool handle. The on-card memory pool handle returned during an STE_REGISTER successful response. (See s96_register)

*csize* Input. Number of bytes to be available in the on-card memory pool for the bufcall to mature (that is, the callback function is called). The number of bytes cannot exceed the size of the registered maximum data buffer size for the pool.

*cfunc* Input. User callback function called when the bufcall matures (that is, the callback function is called). The routine pointer must be a system unit address.

*cparm* Input. Parameter to pass to the `cfunc` user callback function when called.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The bufcall is registered and pending, unless one or more of the following error codes is true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_EINVAL | Invalid *reserved* parameter value. |
| SMI_INVPARM | Invalid *csize* specified.<br>Invalid *cfunc* pointer specified. |
| SMI_TRYLATER | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |
| SMI_TRYALLOC | This return code is an information code indicating that the RadiSys ARTIC960 STREAMS user should retry allocating a buffer from the pool because the RadiSys ARTIC960 STREAMS has detected some additional free memory. The bufcall request was *not* registered. s96_bufcall function *must* be called again. |

### Remarks

Only one outstanding bufcall SKF API can be pending at a time for the same on-card memory pool. If the s96_bufcall function is performed before a previous bufcall SKF API request is completed, the information from the new request is used to override the pending bufcall SKF API request.

To cancel an existing bufcall request, use the s96_unbufcall function.

The SMI_TRYALLOC is generated to account for the asynchronous nature of frees and allocations because the two are done concurrently from different units. Consider the following situation.

> When a user memory allocation fails, perhaps because of the pool's memory shortage, the user calls the s96_bufcall function to wait until memory is available again. But during the time the failed allocation was tried and the s96_bufcall function call was made, the other unit could have freed some buffers. There is the possibility that the allocation would succeed now. In that case, the SMI_TRYALLOC code is returned and the user is expected to retry its allocation. The RadiSys ARTIC960 STREAMS keeps track of frees between each attempt at setting a bufcall and returns this error code every time. So the loop of allocate-fail-bufcall is broken when either the allocation succeeds, or the s96_bufcall succeeds. The scenario is depicted in Figure 7-1.

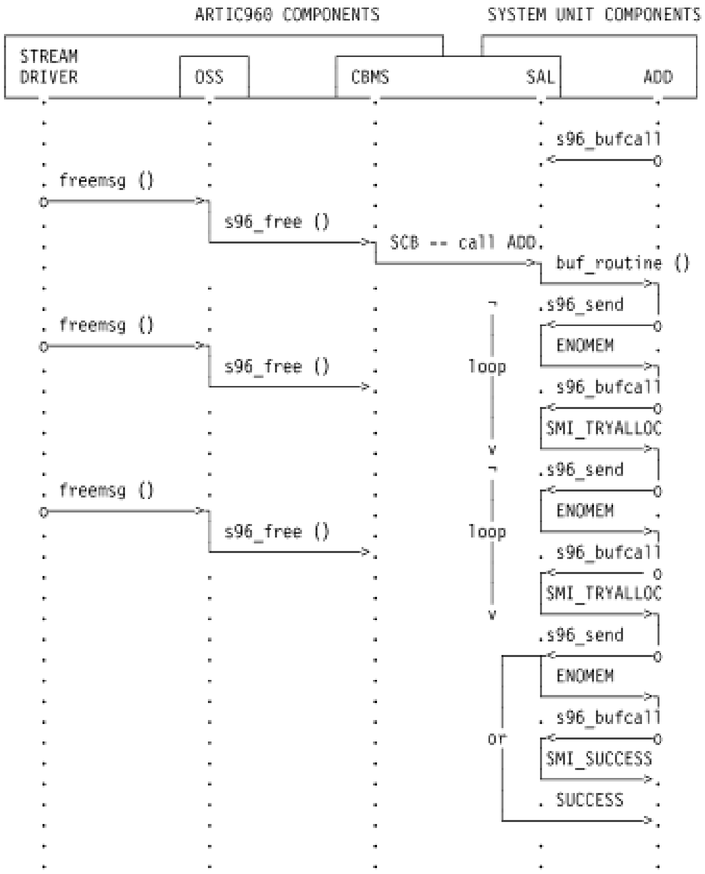

**Figure 7-1. Downstream Flow — SMI_TRYALLOC Situations**

### Response Description

There is no response associated with this request.

## s96_deregister

Deregisters a previously registered on-card memory pool located on the RadiSys ARTIC960 adapter.

### Functional Prototype

```
unsigned long   s96_deregister    (unsigned long          handle,
                                  unsigned long         reserved);
```

### Parameters

*handle* Input. On-card memory pool handle. The memory pool handle returned during an STE_REGISTER successful response. (See s96_register)

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The memory pool is deregistered unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_EINVAL | Invalid *reserved* parameter value. |

### Remarks

The s96_deregister function schedules the deallocation of the on-card memory pool when none of the memory from that pool is in use. The caller can then use only the s96_free function to release the memory in use. (See s96_free for more information.)

### Response Description

The *command* parameter is set to STE_DEREG.

The *arg1* and *arg2* parameters are set to 0.

# s96_expand

Expands the size of an on-card memory pool.

Expands the available amount of bytes in an on-card memory pool with the amount of bytes specified in *rsize*, or up to the maximum pool size, whichever is reached first.

## Functional Prototype

```
unsigned long   s96_expand    (unsigned long          handle,
                               unsigned long          rsize,
                               unsigned long          reserved);
```

## Parameters

*handle* Input. The on-card memory pool handle returned during an STE_REGISTER successful response. (See s96_register.)

*rsize* Input. Amount of bytes to add in the pool.

*reserved* Input. Reserved use by provider. Value must be 0.

## Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

## Error Codes

The on-card memory pool is expanded unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_EINVAL | Invalid *reserved* parameter value. |
| SMI_MAXSIZE | The pool has reached its maximum size; it cannot expand. |
| SMI_OUTOFRESOURCE | Not enough RadiSys ARTIC960 adapter memory available. |

## Remarks

The s96_expand function checks that the pool did not reach its programmed maximum size before expanding it.

The extra allocated size is always rounded up to the next maximum data buffer size specified during a s96_register function.

## Response Description

The *command* parameter is set to STE_EXPAND.

On successful completion, the *arg1* parameter is set to the actual amount of bytes added to the pool. Usually this should be the same amount as specified in the *rsize* request parameter, but the following exceptions may apply.

- An *arg1* value of 0, with the return code SMI_MAXSIZE, signifies that the pool has reached its maximum size and cannot be further expanded.
- If *rsize* specified a size that makes the total pool size greater than its maximum size, the amount of bytes added is reduced to fit the maximum pool size and the *arg1* parameter gives the reduced amount added..

  Because the *rsize* is rounded up to the maximum data buffer size, the amount may have increased the total pool size beyond its maximum size.

- If the request specified a size less than the maximum data buffer size, the size allocated is rounded up to the next maximum data buffer size value.

The *arg2* parameter is set to 0.

## s96_free

Frees an allocated on-card memory pool area.

The SAL users usually do not need to free memory from messages allocated out of the on-card memory pool handle they specify during a s96_send function because the SAL is managing it during the send operation. However, when the s96_send function fails with error code SAL_EAGAIN, the SAL might have allocated the stream message memory out of the on-card memory pool and given the flat RadiSys ARTIC960 adapter memory pointer back to the user. The s96_free function can be used to specifically free the on-card stream message if the user does not want to keep the allocation in between retries.

### Functional Prototype

```
unsigned long   s96_free      (unsigned long          handle,
                               void                   *ptr);
```

### Parameters

*handle* Input. On-card memory pool handle. The memory pool handle returned during an STE_REGISTER successful response (see s96_register ).

*ptr* Input. Flat RadiSys ARTIC960 adapter pointer onto the memory area to free.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The memory is freed unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |

### Remarks

There is no control on memory ownership in order to free the on-card memory area. If the *ptr* is NULL or the *handle* addresses an adapter that has been reset since the on-card memory pool was created, the function returns SMI_SUCCESS but no action is performed..

Unpredictable results may occur in certain circumstances while freeing the same location twice.

### Response Description

There is no response associated with this request.

## s96_info

Queries information about an on-card memory pool.

### Functional Prototype

```
unsigned long   s96_info      (unsigned long         handle,
                               struct CBMS_info      *bptr,
                               unsigned long          reserved);
```

### Parameters

*handle* Input. On-card memory pool handle. The on-card memory pool handle returned during an STE_REGISTER successful response (see s96_register).

*bptr* Input. Information Data Buffer pointer. This is the pointer on the first byte of information data copied. The information data buffer is structured as:

```
struct CBMS_info {
   unsigned long      msize;
   unsigned long      csize;
   unsigned long      cfree;
};
```

where:

*msize* Output. Maximum size, in bytes, of the on-card memory pool. This is the maximum amount to which the pool may expand. See s96_register for more details. This number may not be exactly the same as the one on the s96_register function because the RadiSys ARTIC960 STREAMS may do some rounding to facilitate internal processing. Regardless, it will never be less than the *msize* set in s96_register function.

*csize* Output. Current size, in bytes, of the on-card memory pool. This value gets updated as the pool gets expanded or reorganized. This value can never exceed the *msize*. The difference between these numbers (*msize* – *csize*) is the amount the pool may be expanded. (See *rsize* parameter description at s96_expand.)

*cfree* Output. Current number of bytes currently free in the on-card memory pool. This value gets updated as the pool gets expanded or reorganized. It is also updated whenever allocations and frees are done from the pool. There is no guarantee that an allocation of this *cfree* size will succeed as free memory might be scattered. The success of any allocation depends on these key factors:

- Fragmentation of the on-card memory pool.
- In any case the maximum size of an allocation never exceeds the size of the maximum data buffer lengths. But this number may be greater than the *flblen* size requested in s96_register function call.
- As with any memory management scheme, there is a small fixed overhead per allocation (8 bytes). Even if only one byte is requested, a larger amount of memory gets subtracted from this amount.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The information is returned regarding the memory pool unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_EINVAL | Invalid *reserved* parameter value. |
| SAL_EFAULT | The *bptr* pointer is invalid. |

### Remarks

None.

### Response Description

There is no response associated with this request.

## s96_register

Registers a shared memory pool on the RadiSys ARTIC960 adapter.

Messages sent by the s96_send are stored in the memory pool. Before s96_send is called, the shared memory must be expanded using the s96_expand function call.

### Functional Prototype

```
unsigned long   s96_register    (int                    cardnum,
                                  unsigned long          msize,
                                  unsigned long          flblen,
                                  unsigned long          memo,
                                  unsigned long          reserved);
```

### Parameters

*cardnum* Input. The logical RadiSys ARTIC960 adapter number where to register the shared memory pool. Valid adapter numbers range from 0–15.

*msize* Input. Maximum pool size (in bytes). The pool can expand to the size specified by this parameter.

*flblen* Input. Maximum data buffer size (in bytes). This parameter:

- Determines the maximum length a buffer can have for downstream data transmission.
- May be rounded upward to facilitate its internal processing.
- Determines the multiple by which the pool will grow while performing a s96_expand function call. If a value of 0 is passed, a default value, equal to the OSS's MAXBLOCKLEN parameter value, is set for the pool. (See page 10 for a description of the MAXBLOCKLEN parameter.)

*memo* Input. Correlation value. This value is provided during asynchronous notification of memory responses' events. The content of this variable is implementation defined, but is primarily intended as a pointer or an index that would aid the device driver in locating instance-specific information.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The shared memory pool is registered unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVPARM | Invalid parameters. |
| SMI_OUTOFRESOURCE | Not enough RadiSys ARTIC960 adapter memory available. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |

| | |
|---|---|
| SMI_TRYLATER | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |

### Remarks

The shared memory pool must be registered before any other memory function can be performed.

More than one shared memory pool is allowed.

### Response Description

The *command* parameter is set to STE_REGISTER.

On successful completion, the *arg1* parameter is the shared memory pool handle for the new pool registered. This handle must be used whenever a subsequent command applies to this memory pool. The *arg2* parameter is 0.

## s96_reorg

Reorganizes the on-card memory pool.

When the on-card memory pool has been extensively used, especially performing small allocations and frees, the RadiSys ARTIC960 STREAMS's scan for a suited data buffer within the maximum data buffer pool may take longer. Reorganizing the pool speeds up the process.

### Functional Prototype

```
unsigned long   s96_reorg     (unsigned long          handle,
                                unsigned long          action,
                                unsigned long          reserved);
```

### Parameters

*handle* Input. On-card memory pool handle. The memory pool handle returned during an STE_REGISTER successful response (see s96_deregister).

*action* Input. Optional selective actions performed by the RadiSys ARTIC960 STREAMS if requested by the caller.

- RET_AVAIL

  Instructs the RadiSys ARTIC960 STREAMS to deallocate any on-card memory pool eligible for deallocation. An eligible on-card memory pool is one with no outstanding suballocation. The remaining on-card memory pool is then re-ordered to increase chances for a successful allocation.

*reserved* Input. Reserved use by provider. Value must be 0.

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The on-card memory pool is expanded unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |
| SAL_EINVAL | Invalid *reserved* parameter value. |
| SMI_INVPARAM | Invalid action bits specified. |

### Remarks

None.

### Response Description

The *command* parameter is set to STE_REORG.

The *arg1* and *arg2* parameters are set to 0.

## s96_unbufcall

Deactivates a previous bufcall request.

The last successful s96_bufcall request is canceled.

### Functional Prototype

```
unsigned long   s96_unbufcall   (unsigned long          handle,
                                  unsigned long          reserved);
```

### Parameters

| | |
|---|---|
| *handle* | Input. The on-card memory pool handle returned during an STE_REGISTER successful response. (See s96_register.) |
| *reserved* | Input. Reserved use by provider. Value must be 0. |

### Returns

Output. On successful handling of the request, a value of 0 is returned. An error code other than 0 indicates the error.

### Error Codes

The bufcall is deregistered and pending unless one or more of the following error codes are true:

| | |
|---|---|
| SMI_INVHANDLE | Invalid on-card memory pool handle. |
| SAL_ENOCONNECT | Unable to communicate with the RadiSys ARTIC960 STREAMS. |

| | |
|---|---|
| SAL_EINVAL | Invalid *reserved* parameter value. |
| SMI_TRYLATER | Temporarily unable to send the request to the RadiSys ARTIC960 adapter. |

### Remarks

An s96_unbufcall function is ignored when there is no bufcall request in progress. Because there can be only one outstanding bufcall request at a time per entity, the unbufcall request cancels the current active bufcall request.

### Response Description

There is no response associated with this function.

## Response Codes

Response codes are command blocks sent by the OSS that are not responses to a previous request triggered by the ADD. Response codes are delivered through the s96_resphandler asynchronous response handler function. See Response Handler on page 74 for information on the SAL's use of the s96_resphandler function.

For all response codes, the *memo* parameter is the value passed during the s96_open function for the stream unless otherwise specified.

The following are the response codes and their descriptions. There are no specific error codes defined.

STE_XRECEIVE

Indicates the reception of a stream message from the RadiSys ARTIC960 STREAMS.

The *command* parameter is set to STE_XRECEIVE.

The *arg1* parameter is a pointer to the first message block structure (mblk_t) of the message. The message block structure contains a pointer to one or more data block structure(s) dblk_t. The message blocks and data block memory are allocated using the system unit's operating system stream environment. It is the application's (or, at least, the stream head's) responsibility to free the memory associated with the stream message received.

The *arg2* parameter is 0.

STE_STOPXMIT

Instructs the ADD to stop sending messages downstream.

The *command* parameter is set to STE_STOPXMIT.

The on-card stream head write service queue becomes full when it receives messages from the system unit and cannot deliver them successfully to the next STREAMS-based module/driver, causing its predefined high water mark level to be exceeded.

Both parameters, *arg1* and *arg2* are zero (0).

STE_STARTXMIT

Instructs the ADD to restart sending messages downstream.

The *command* parameter is set to STE_STARTXMIT.

The on-card stream head write service queue becomes available to process messages from the system unit after its predefined low watermark level is reached.

Both parameters, *arg1* and *arg2*, are 0.

STE_CONNECT

Informs the ADD about the status of communication with an RadiSys ARTIC960 STREAMS.

The *command* parameter is set to STE_CONNECT.

Each time the status of one of the RadiSys ARTIC960 adapters hosting the RadiSys ARTIC960 STREAMS *changes*, the new state of the RadiSys ARTIC960 adapter is reported to the ADD. The following states are defined.

S96_COMM_UP

The state reported when SCB Pipes are configured and contact has been established successfully between the SAL and the RadiSys ARTIC960 STREAMS. In this state, the ADD can communicate with the RadiSys ARTIC960 adapter using all verbs from the SAL API.

S96_COMM_DOWN

The state reported whenever communication between the SAL and an RadiSys ARTIC960 STREAMS is broken. Reasons might be:

- The card is reset
- A terminal error was reported
- SCB pipes get unconfigured
- SCB pipes access timed out
- Any other error occurred in the RadiSys ARTIC960 STREAMS.

S96_COMM_XON

The state reported whenever the Global Communication Channel between the SAL and the RadiSys ARTIC960 STREAMS allows non-high-priority messages to be sent downstream again (flow control situation ends). The ADD should retry sending data downstream for any of its opened streams that have previously been flow controlled (STE_STOPXMIT received for the stream).

The *arg1* parameter contains the logical card number for which status is being reported.

The *arg2* parameter contains the card status..

The *memo* parameter is set to 0.

# Log Device Driver

The log driver is an RadiSys ARTIC960 STREAMS software device driver that provides an interface for the RadiSys ARTIC960 STREAMS error and event-tracing processes. The log driver presents the following separate interfaces.

- strlog() SKF API from within a STREAMS-based module/driver in the RadiSys ARTIC960 adapter. The strlog() SKF API is described in the UNIX SVR4 STREAMS documentation.
- A subset of ioctl operations and RadiSys ARTIC960 STREAMS messages for interaction with a user-level error logger and/or tracer.

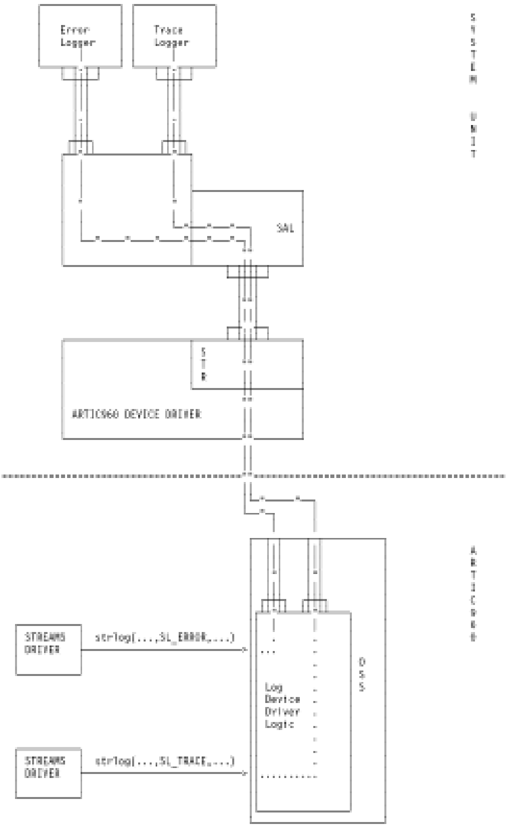

Figure 7-2. Error and Trace Loggers

## User-Level Access.

**Programming Note**
All references to C defines and structures can be found in the ***sys/Ostrlog.h*** file. This file is shared by both the system unit’s user-level process and the RadiSys ARTIC960 adapter STREAMS-based module/driver, with either the RIC_AIX_RS6000 or RIC_KERNEL define to be enabled.

The log device driver gets automatically installed as a STREAMS-based driver when the OSS is loaded in the RadiSys ARTIC960 adapter.

The log device driver is opened using the clone interface for the device name **riclg**. See s96_open for details on open parameters. Each open of the **riclg** driver obtains a separate stream to this driver, which is capable of acting as an error or trace logger. To select which it will be, the user-level process uses a defined I_STR ioctl immediately after the stream has been opened with the log driver.

error logger

The I_STR operation has an ic_cmd field value of I_ERRLOG with no additional data.

trace logger

The I_STR operation has an ic_cmd field value of I_TRCLOG with additional data specifying selected criteria the log record should meet in order to be reported to the trace logger.

- The data buffer is an array of one or more card_trace_ids_t structures. Each cell specifies mid**,** sid and level fields from which messages are accepted.
- The strlog() SKF API accepts records whose values in the mid**,** sid and level fields match the selection made through the card_trace_ids_t structures.
- A value of –1 in any of the fields of the card_trace_ids_t structure indicates that any value is accepted for that field.

At most, one error logger and one trace logger can be active at a time. Once the logger process has identified itself using the ioctl operation described previously, the log driver begins sending messages, subject to the restrictions previously selected. The user-level process receives those log messages through the getmsg() system call. The control part of the message contains a card_log_ctl_t structure that specifies:

*mid* Module ID

*sid* Subsystem ID

*level* Log level

*flags* Log flags

*ttime* Time in ticks since RadiSys ARTIC960 adapter's reset when the log message was submitted (with HZ = 200 for the RadiSys ARTIC960 adapter).

*ltime* Time in seconds since January 1, 1970, when the log message was submitted..

In order for the time to be reported accurately by the RadiSys ARTIC960 adapter, the time-of-day timer must be enabled. Refer to the *RadiSys ARTIC960 Programmer's Reference* for information on the **ricload** utility.

*seq_no* Sequence number for the log message

The data part of the message contains the *formatted*, null-terminated string with its accompanied arguments passed by the strlog() SKF API.

The following error codes are returned on completion of the ioctl operation by the log driver.

| | |
|---|---|
| ENXIO | A logging process of the given operation type (I_ERRLOG/ I_TRCLOG) already exists. |
| ENXIO | The I_TRCLOG operation does not contain any card_trace_ids_t structures. |
| ENOSR | Maximum number of specific sids per mid reached (64 sids maximum). |
| EINVAL | Invalid operation code. |
| EINVAL | Unsupported message type. |

## Kernel-Level Access

Refer to the **sys/Ostrlog.h** file for values to use for level and flags fields of the strlog() SKF API.

### SL_HEXA Addition

A flag, SL_HEXA, has been added to existing standard UNIX SVR3/4 flags. This new flag gives the ability to report hexadecimal strings of bytes through the same interface, thus enabling communication's frames to be reported to the trace logger. SL_HEXA can only be specified in conjunction with SL_TRACE and *must* have the strlog()'s *arg1* parameter set to the actual hexadecimal string's length. **fmt** then contains the pointer on that string.

ONLOGARGS defines the maximum number of variable arguments during a strlog() SKF API call. The default is 6.

OLOGMSGSZ defines the maximum length (in bytes) for the log data portion (formatted string). The default is 128.

Log messages received by the user-level logger through the getmsg() system call are little-endian encoded. It is the responsibility of the user-level logger to transform them into big-endian format for AIX.

# Error Codes

Error code values returned by SAL stream and memory function commands and response handler are derived from UNIX System V **errno** values. For an ADD running under a UNIX-type operating system, these error codes can be masked to clear the high order bit (0x80000000) and used as if they were UNIX System V *errno* values.

The ANSI conformance defines *errno's* by name and not by value, which indicates that different operating systems might have the same name defined with two different values. For the SAL, applicable values are the ones listed in the error code tables following each function description.

**Table 7-5. Error Codes**

| Name | Value |
|---|---|
| SAL_EINVAL | 0x80000016 |
| SAL_ENAMETOOLONG | 0x80000056 |
| SAL_EAGAIN | 0x8000000b |
| SAL_EMFILE | 0x80000018 |
| SAL_ENXIO | 0x80000006 |
| SAL_ENOSR | 0x80000076 |
| SAL_EBADF | 0x80000009 |
| SAL_EINTR | 0x80000004 |
| SAL_EFAULT | 0x8000000e |
| SAL_ENOMEM | 0x8000000c |
| SAL_ETIME | 0x80000077 |
| SAL_ENOCONNECT | 0x80000032 |
| SAL_ENOENT | 0x80000002 |
| SMI_INVPARM | 0x80000016 |
| SMI_OUTOFRESOURCE | 0x8000000c |
| SMI_TRYALLOC | 0x80000055 |
| SMI_TRYLATER | 0x8000000b |
| SMI_INVHANDLE | 0x80000009 |
| SMI_MAXSIZE | 0x80000022 |

**Table 7-6. Additional Error Codes Returned by OS/2 APIs**

| Name | Value |
|---|---|
| SAL_ERR_NO_CARD | 0x80001001 |
| SAL_ERR_ALLOC_GDT | 0x80001002 |
| SAL_ERR_SETTIMER | 0x80001003 |
| SAL_ERR_MEMPOOL_INIT | 0x80001004 |
| SAL_ERR_COMMON_INIT | 0x80001005 |

Table 7-7. Additional Error Codes Returned by Windows NT APIs

| Name | Value |
|---|---|
| SAL_ERR_MEMSPINLOCK | 0x80001001 |
| SAL_ERR_NONPAGED_MEM | 0x8000100d |
| SAL_ERR_DEV_OBJ_PTR | x8000100f |
| SAL_ERR_ALLOCIRP | 0x80001010 |
| SAL_ERR_IOCALLDRIVER | 0x80001011 |
| SAL_ERR_IDCEVENT | 0x80001012 |
| SAL_ERR_INITOSS | 0x80001015 |
| SAL_ERR_INITMEM | 0x80001016 |
| SAL_ERR_NULLPTR | 0x80001007 |
| SAL_ERR_MSGLINKED | 0x80001008 |

# OS/2-Specific Functions

The following list of functions have been developed for the OS/2 SAL, in addition to the Streams Functions and memory functions described in Stream Functions on page 79 and Memory Functions on page 92.

| Function | Description |
|---|---|
| s96_freemsg | This function frees a streams message |
| s96_idc_init | Initializes the SAL OS/2 interface with the OSS |
| s96_os2_init | Initializes the SAL OS/2 interface |

## s96_freemsg

This function frees a streams message and returns it to the internal SAL memory pool.

### Functional Prototype

```
unsigned long s96_freemsg (mblk_t *mp)
```

### Parameters

*mp* Input. Pointer to streams message block.

### Remarks

Other than the b_rptr and b_wptr, no other internal field of the mblk_t structure must be modified by the user of this function. If the b_next field of the structure is changed, it should be made NULL before calling this function.

### Error Codes

Any return code other than 0 is an error. The function could fail with the following error codes.

*3* The message pointer (parameter mp) is NULL.

*5* The internal data structure of the message being freed is corrupted. The b_next field of the message should be NULL.

## s96_idc_init

Initializes the SAL OS/2 interface with the OSS by issuing interdevice driver communications (IDC) calls to the RadiSys ARTIC960 device driver.

### Functional Prototype

```
unsigned long s96_idc_init ();
```

### Parameters

None.

### Remarks

The s96_idc_init function must be called during the INITIALIZATION COMPLETE event, during which the OS/2 kernel calls the driver's strategy routine with this function code. The kernel allows the device drivers to set up any IDC interfaces at this point.

### Error Codes

Any return code other than 0 is an error. The function could fail with the following error codes.

| | |
|---|---|
| SAL_ERR_MEMPOOL_INIT | The SAL internal memory pool could not be allocated and/or formatted. |
| SAL_ERR_COMMON_INIT | The SAL failed to set up initial communication with the card. |

## s96_os2_init

Initializes the SAL OS/2 interface by taking up appropriate resources that are required in the KERNEL operating mode of the driver.

### Functional Prototype

```
unsigned long s96_os2_init ()
```

### Parameters

None.

### Remarks

The s96_os2_init function must be called during the INIT mode of the OS/2 driver. It should be noted that the call to this function is synchronous and the return code is available immediately.

### Error Codes

Any return code other than 0 is an error. The function could fail with the following error codes.

| | |
|---|---|
| SAL_ERR_NO_CARD | No RadiSys ARTIC960 card is detected in the system. |

| | |
|---|---|
| SAL_ERR_ALLOC_GDT | Global Descriptor Table entries for memory management could not be allocated. |
| SAL_ERR_SETTIMER | The timer for the SAL internal timer functions could not be started. |

# Windows NT-Specific Functions

The following list of functions have been developed for the Windows NT SAL, in addition to the streams functions and memory functions described in Stream Functions on page 79 and Memory Functions on page 92.

| Call | Description |
|---|---|
| s96_freemsg | Frees up a streams message block and all its related structures. |
| s96_nt_getcard | Returns the logical card number for each of the card installed in the system. |
| s96_nt_haltsys | Does the cleanup and resource allocation for the SAL. |
| s96_nt_initsys | Does the global initialization and allocation of resources required by the SAL. |
| s96_nt_timeout | Registers a timer handler. |
| s96_nt_untimeout | Cancels a previously registered timeout. |

## s96_freemsg

Frees a streams message block and all its related structures.

This function must be called by the driver for all the messages that it gets upstream by way of the SAL response handler. This function frees up the memory in the SAL pool and makes it available for other messages being delivered by the RadiSys ARTIC960 driver to the SAL.

### Functional Prototype

```
unsigned long s96_freemsg (mblk_t *mp)
```

### Parameters

| | |
|---|---|
| *mp* | Input. Pointer to streams message block. |

### Remarks

None.

### Error Codes

Any return code other than 0 is an error. The function could fail with the following error codes.

| | |
|---|---|
| SAL_ERR_NULLPTR | The message pointer passed is NULL. |
| SAL_ERR_MSGLINKED | This message has a link to another message and, therefore, cannot be freed. |

## s96_nt_getcard

Returns the logical card number for each of the cards installed in the system.

### Functional Prototype

```
int s96_nt_getcard (int PrevCardNumber);
```

### Parameters

*PrevCardNumber*

Input. This is the card number returned by the function in the previous invocation. For the first invocation, it should be GET_FIRST_CARD.

### Returns

Returns the logical card number, starting from 0, onward. Returns NO_MORE_CARD when all cards have been enumerated.

### Remarks

This call must be made from the DriverEntry() of the Windows NT kernel driver after it has made the call to s96_nt_initsys().

### Error Codes

None.

## s96_nt_haltsys

Does the cleanup and resource deallocation for the SAL.

### Functional Prototype

```
void s96_nt_haltsys ()
```

### Parameters

None.

### Remarks

This function must be called by the Windows NT kernel driver at the time of driver unload.

### Error Codes

None.

## s96_nt_initsys

Does the global initialization and allocation of resources required by the SAL for all the RadiSys ARTIC960 adapters in the system.

### Functional Prototype

```
unsigned long s96_nt_initsys ()
```

### Parameters

None.

### Remarks

This function must be called by the Windows NT kernel driver when it gets called at its DriverEntry() entry point.

### Error Codes

Any return code other than 0 is an error. The function could fail with the following error codes.

| | |
|---|---|
| SAL_ERR_MEMSPINLOCK | Failure to get memory for spin locks. |
| SAL_ERR_NONPAGED_MEM | Failure to allocate non-paged memory pool. |
| SAL_ERR_DEV_OBJ_PTR | Failure to get the device object pointer to the RadiSys ARTIC960 driver. |
| SAL_ERR_ALLOCIRP | Failure to allocate the I/O Request Packet (IRP) for getting RadiSys ARTIC960 function pointers. |
| SAL_ERR_IOCALLDRIVER | Failure to issue IoCallDriver() to the RadiSys ARTIC960 driver. |
| SAL_ERR_IDCEVENT | Error waiting for event to get interdevice driver communications (IDC) function pointers. |
| SAL_ERR_INITOSS | Failure to initialize and bind with the On-card STREAMS Subsystem (OSS). |
| SAL_ERR_INITMEM | Failure to initialize memory for received messages. |

## s96_nt_timeout

Registers a timer handler to be called after the expiration of the specified timeout period.

### Functional Prototype

```
int s96_nt_timeout ( PFNRV func, ULONG arg, ULONG msecs)
```

### Parameters

| | |
|---|---|
| *func* | Input. Pointer to the function to be called at timeout. |
| *arg* | Input. Argument to be passed to the function called at timeout. |
| *msecs* | Input. Timeout period expressed in 1000-millisecond interval. |

### Remarks

None.

### Error Codes

Any return code other than 0 is the correlation value, which is used to cancel the timeout using the s96_nt_untimeout() function. Failure is indicated by returning a –1.

| | |
|---|---|
| *>0* | Timeout correlation value. |
| *–1* | Failure to issue a timeout. |

## s96_nt_untimeout

Cancels a previously registered timeout.

### Functional Prototype

```
void s96_nt_untimeout ( unsigned long corr)
```

### Parameters

| | |
|---|---|
| *corr* | Input. Correlation value returned previously by s96_nt_timeout(). |

### Remarks

None.

### Error Codes

None.

# Notices

A

This information was developed for products and services offered in the U.S.A. RadiSys Corporation may not offer the products, services, or features discussed in this document in other countries. Consult your local RadiSys representative for information on the products and services currently available in your area. Any reference to a RadiSys product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any RadiSys intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-RadiSys product, program, or service.

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

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(561) 454-3200

# Index