System Services Interface

Service Definition

1111001 3.12a

System Services Interface

Service Definition

1111001 3.12a

Radisys Corporation

9450 Carroll Park Drive San Diego, CA 92121-2256

Phone: *+1 (858) 882-8800*

Fax: *+1 (858) 777-3389*

Web: *http://www.radisys.com*

System Services Interface
Service Definition
1111001 3.12a

Contributors: Continuous Computing Development Team, Naveen D’cruz.

Printed in U.S.A.

Contents

Figures

Tables

Preface

Objective

This document provides a detailed description of the services provided by the System Services Interface software designed by Continuous Computing Corporation.

Audience

The readers of this document are assumed to be engineers with a knowledge of telecommunication protocols, specifically Operating Systems.

Document Organization

This document is organized into the following sections:

Section	Description
1 Introduction	Describes the overview of the product.
2 Environment	Describes the assumptions about the software environment for operation of the System Services Interface software.
3 Trillium Advanced Portability Architecture (TAPA)	Describes the TAPA architectural and coding standards.
4 System Services	Describes all the functions defined at the System Services Interface.
5 Usage	Describes the usage of the System Services Interface.
6 Porting Issues	Describes the factors that must be considered during the portation of the System Services Interface.

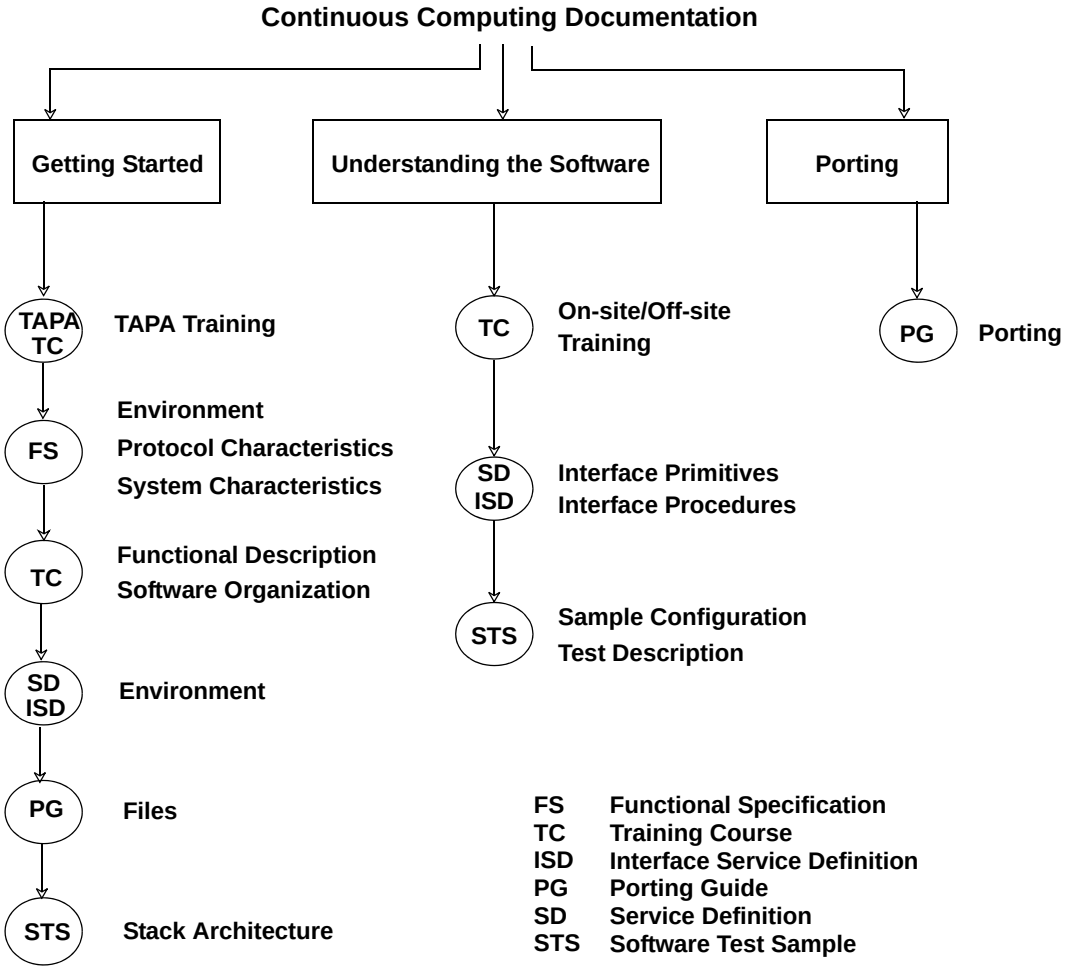
Notations

This table displays the notations used in this document.

Notation	Explanation	Examples
Arial	**Title s**	**1.1 Title**
Book Antiqua	Body text	This is body text.
Bold	**Notes**	**Note:** This is an example of a note. The text is indented.
ALL CAPS	CONDITIONS, MESSAGES	TRUE or FALSE CONNECT ACK
Italics	*Document names emphasis.*	*AMT Service Definition.* *This adds emphasis.*
Courier New Bold	**Code Filenames, pathnames**	**PUBLIC S16 AmMiLamCfgReq(pst, cfg) Pst *pst; AmMngmt *cfg;**

Using Continuous Computing® Documentation

The following figure shows the various user approaches to using the software documentation. First time users must read the documents under the **Getting Started** column, where important sections and subsections are listed to the right of each document. For users familiar with the documentation, but who need to look up certain points concerning software use, **Understanding the Software** column is suggested. The **Porting** column is for users familiar with Trillium software and related telecommunications protocols and wish to install the software immediately onto their development environments.



Release History

This table lists the history of changes in successive revisions of this document.

Version	Date	Author (s)	Description
3.12a	October 10, 2011	Naveen Dcruz H	Addendum release for Radisys logo and template upgrade.
3.11a	April 04, 2009	Sharat Chandra	Addendum release. Conforms to SSI licensing option.
3.1	October 04, 2008	Chetan Hebli	Conforms to SSI multi-core enhancements release.
1.7	March 14, 2005	Sripriya Kapali	Updated with new logo.
1.6	March 24, 1998	aa ada	• Added F/Q System Service. • Added **`SAddMsgRef`** System Service.
1.5	August 6, 1996	ft.	• Added new System Services Interface primitives. • Updated environment descriptions.

1

Introduction

This document provides the definition of the System Services Interface used by all software products designed by Continuous Computing Corporation. This interface encapsulates all operating system dependencies of the portable software, thereby enabling the C source code to be compiled to run under any operating system and system architecture.

This document must be typically read in conjunction with the Service Definition of a particular product.

The following abbreviations are used in this document:

Abbreviation	Description
ANSI	American National Standards Institute
CCITT	Consultative Committee on International Telephone and Telegraph
ISO	International Standards Organization
MOS	Multiprocessor Operating System
OSI	Open Systems Interconnection
SAP	Service Access Point
TAPA	Trillium Advanced Portability Architecture

For a list of commonly used terms, refer to the Engineering Glossary (part number PREN026) at http://www.ccpu.com/search/glossary/

2

Environment

This section describes the assumptions about the environment in which the software product is designed to operate.

2.1 Processors

The system may consist of one or more processors. Processors are identified by processor identifiers (**`ProcId`**) that are globally unique.

A special value (**`PROCIDNC`**) is used to denote an illegal processor ID.

Processors may be associated with memory regions that are private to one processor and shared with other processors.

Processors may communicate with each other either through shared memory regions (through queues) or communications channels (through any bilaterally defined protocol).

2.2 Tasks and Activation Functions

A task is an invocation of a Trillium product.

An activation function is a schedule able entry point within a task. There are usually many activation functions within a task, as described below.

A system process may consist of one or more tasks. A processor may run one or more processes.

Tasks are identified by an entity ID and an instance ID.

An entity ID (**`Ent)`** is a reference number used to functionally distinguish the Trillium products (for example, protocol type). Common entity IDs are defined in **`ssi.h`** file. A special value (**`ENTNC`**) is used to denote an illegal entity ID.

An instance ID (**`Inst)`** is a sequence number used to distinguish among multiple instances of the same entity (for example, two copies of a protocol layer).

2.2.1 Driver Tasks

The System Services Interface provides for the concept of "Driver Tasks." In this discussion, driver tasks relate to special tasks registered with system services (refer to **`SRegDrvrTsk`**) that provide communications links for moving messages (through loosely coupled interfaces) between separate stack implementations that cross arbitrary boundaries (that is, distributed architectures). Driver tasks shuttle messages between two or more distinct stacks. This definition differs from that of a protocol-related driver task (physical layer implementation).

2.2.2 Inter-Task Communication

Tasks communicate with each other through direct function calls (tight coupling) or through message passing (loose coupling). The former is accomplished through shared memory (function stack) while the latter may be done either through shared memory (message queues) or through a communication channel. Tasks do not share any global data structures. Later sections of this document describe tasks in more detail.

2.2.3 Activation Function Types

There are four types of activation functions, as described in Section 2.2.3.1 to Section 2.2.3.4.

Activation function types are defined in **`ssi.h`** file. No type is explicitly defined for initialization and timer functions, since they are handled using explicitly different function interfaces.

The terms "task" and "activation function" are used interchangeably when the semantics are clear; that is, since there is exactly one message activation function for a task, it may also be referred to as the message activation task.

2.2.3.1 Initialization Activation Function

This function is the entry point for a task to initialize its global variables. There is a single such function per task. This function is scheduled exactly once, before the task begins its operations.

2.2.3.2 Normal Message Activation Function

This activation function is identified by the global symbol **`TTNORM`**, which is defined in **`ssi.h.`** This function handles all messages sent to a task. There is a single such function per task. This function is scheduled whenever another task sends a message to this task through its loosely coupled interfaces.

2.2.3.3 Timer Activation Function

This activation function is used to receive periodic timer ticks from the system services so that the task can manage its internal timers. There may be one or more such functions per task. Typically, there is one such function for each different timer resolution required by the task. This function is scheduled periodically by the system services at pre-specified intervals.

2.2.3.4 Permanent Activation Function

This activation function is identified by the global symbol **`TTPERM`**, which is defined in **`ssi.h`**. This function is scheduled at irregular intervals to perform periodic functions (such as monitoring a test). There is no fixed period for scheduling this function and it is typically activated when no other function can be activated. There can be one or more such functions per task.

2.2.3.5 Activation Function Scheduling

Activation functions are scheduled depending on their type. For example, timer functions are scheduled at pre-specified intervals, and message handling functions are scheduled whenever there is a message for a task. Scheduling of the message handling functions can be preemptive or non preemptive, based on the underlying operating system or System Service Provider (SSP). However, because all Trillium products are non-reentrant, care must be taken in the development of SSPs to account for this.

2.3 Memory Regions, Pools and Buffers

A memory region is typically a physical block of memory associated with one or more processors.

Memory regions are identified by a region ID (**`Region`**). A special value (**`REGNC`**) is used to denote an illegal region ID.

Memory regions may be shared or private. Shared memory regions are accessible by two or more processors. Private memory regions are under exclusive control of a single processor.

The distinction between shared and private memory regions is related only to the logical organization of the memory and not necessarily to its physical organization.

For efficient management, memory regions may be divided into one or more memory pools.

Memory pools are identified by a pool ID (Pool). A special value (**`POOLNC`**) is used to denote an illegal region ID.

The structure of pools is system-dependent. Pools may consist of contiguous byte arrays that can be broken into smaller byte arrays or of fixed-size byte arrays that are linked together.

2.3.1 Memory Pool Types

There are two types of pools, as follows.

2.3.1.1 Static Memory Pools

Static pools are used for allocating static buffers (described in Section 2.3.2.1). The task manages allocation and deallocation of buffers from this pool as well as the contents of the allocated buffers. Each task is returned a single static pool ID when it requests its static memory from system services (refer to Section 4.14.5.1).

2.3.1.2 Dynamic Memory Pools

Dynamic pools are used for allocating dynamic buffers (described in Section 2.3.2.2) to build inter-task messages. A different dynamic pool may be assigned to each Service Access Point (SAP) used by a task. SAPs are described in more detail in Section 3.1.5. The allocation and deallocation of buffers from these pools, as well as the size of the pools, are hidden from the tasks and are managed by system services.

2.3.2 Memory Buffer Types

Memory buffers are the unit of allocation of memory from memory pools to tasks.

There are two types of buffers, as follows.

2.3.2.1 Static Memory Buffers

Static buffers are allocated from static pools. They are used transparently by the tasks for overlaying (dynamic) data structures that the tasks require.

2.3.2.2 Dynamic Memory Buffers

Dynamic buffers are allocated from dynamic pools. They are used by system services to build messages (described in Section 2.4). Consequently, they may need control overhead. For example, to chain multiple buffers to form a single message.

2.4 Messages

A message is a data abstraction that allows information to be sent between tasks. This information can be a protocol SDU or an interface primitive (at a loosely coupled interface).

Logically, a message is an ordered sequence of bytes. The maximum size of a message is determined by the maximum size of a message expected to be transmitted or received by the protocol layer.

Because message structure is invisible to the protocol layers, message management is performed entirely by system services. A dummy message type (**`Buffer`**) is defined in **`ssi.x`** file.

Messages may consist of a contiguous byte array with an associated length of information; or they may consist of a linked list of fixed-size arrays (dynamic buffers); or they may have some other implementation.

Typically, system services needs to maintain some header information with each message such as message sender, receiver, priority, and whether a message is being shared (referenced by more than one user) or not. The contents of the data part of a message are bilaterally defined between the communicating tasks.

2.4.1 Message Priority

In a priority scheduling system, messages may be sent at different priorities.

The following priorities are defined in **ssi.h**:

Table 2-1: Message Priority

Name	Priority
PRIOR0	Priority Zero (Highest).
PRIOR1	Priority One.
PRIOR2	Priority Two.
PRIOR3	Priority Three (Lowest).
PRIORNC	Priority Not Configured.

The layers are configured per SAP with what priority to use when sending messages to upper or lower layers (through loosely coupled interfaces). How priorities are used depends on the System Services Interface implementation.

2.4.2 Message Routing

Messages between communicating tasks may carry routing information that is relevant to getting the messages to the destination task occurring within the system. Such routing information has no correlation to any protocol routing mechanisms.

One instance in which such routing is needed occurs when the sending task wants to broadcast a message to multiple tasks; this may happen when a destination protocol entity has multiple instances running on different processors in a fault tolerant lock-step mode.

The following routes are defined in **ssi.h** file.

Table 2-2: Message Routing

Route	Description
RTESPEC	Route to a specific instance.
RTEFRST	Route to the first available instance.
RTEALL	Route to all instances.
RTENC	Route not configured.

The layers are configured per SAP with what route to use when sending messages to upper or lower layers (through loosely coupled interfaces). How routes are used depends on the System Services Interface implementation.

2.4.3 Message Sharing

A message may be shared by two or more users to try and avoid message duplication, thereby improve the efficiency of the message handling functions. The users of a particular shared message may be within a single task or split across multiple tasks. For the purpose of indicating that a message is being shared between different users, a reference count may be assigned to a message. This reference count indicates the number of users sharing the message. Once a message is marked as being shared, any modifications to the data contents of the message by a particular user must not be reflected across to the other users of the message. This may warrant that the message be duplicated and is an implementation dependent issue.

2.5 Queues

A queue is a data abstraction that allows messages to be stored in an ordered fashion. Typically, a queue is used by a task to retain messages until such time as they are either passed to another task or returned to memory.

The maximum size of a queue is determined by the maximum number of messages expected to be held by the protocol layer at any instant in time. This value may be dependent on protocol window sizes, processing time, and other factors.

The structure of queues is invisible to the protocol layers, so queue management is done entirely by system services. A dummy queue type is defined in **`ssi.x`** (**`Queue`**) file.

Queues may be implemented as a contiguous message pointer array accessed through indexes as messages that are linked together and accessed by a combination of chain pointers or other structures; or they may have some other implementation.

3

Trillium Advanced Portability Architecture (TAPA)

All Trillium products can be described in terms of the Trillium Advanced Portability Architecture® (TAPA®), a set of architectural and coding standards.

3.1 Layer Interfaces

A Trillium product can be visualized as a single box in the center surrounded by four outer boxes. The four outer boxes represent other software to which the Trillium product may be connected.

The separation between the center box and outer boxes represents the interfaces across which the Trillium product interacts with the other software. Each interface consists of a well defined set of C function calls. These interfaces are now described in more detail.

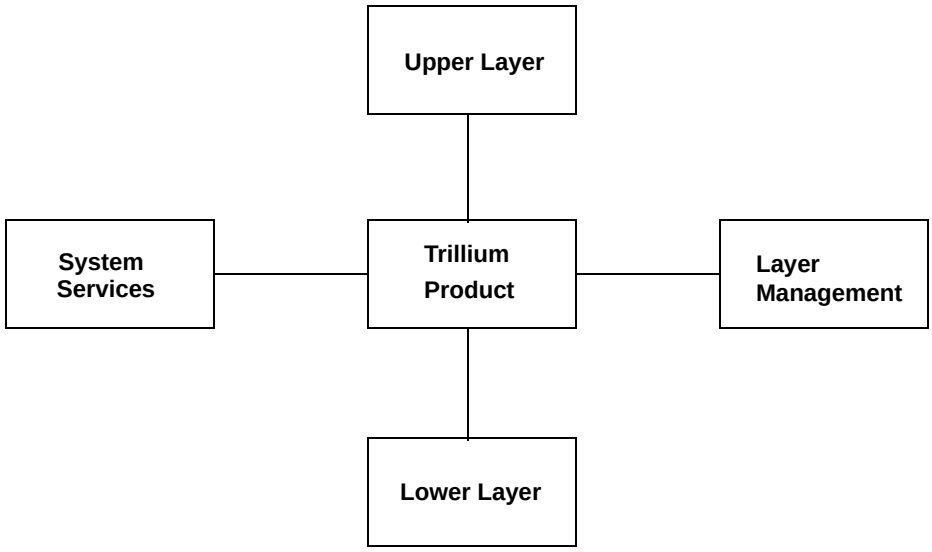


Figure 3-1: TAPA Architecture

3.1.1 Upper Layer Interface

The upper layer interface provides the functions required by the Trillium product to communicate with the upper protocol layer (service user). Typical functions are described in Table 3-1:

Table 3-1: Upper Layer Interface

Function	Description
Bind/Unbind	Registers and de-registers service user with service provider.
Connect/ Disconnect	Initiates peer-to-peer connection and disconnection procedures.
Data Transfer	Initiates peer-to-peer data transfer procedures.
Reset	Initiates peer-to-peer reset procedures.
Flow Control	Initiates flow control between service user and service provider.

The upper layer interface may be different for each product and may not exist for some products. This interface is described within the Service Definition of the appropriate product.

The upper interface of one product (service provider) is designed to match the lower interface of the corresponding product (service user).

3.1.2 Lower Layer Interface

The lower layer interface provides the functions required by the Trillium product to communicate with the lower protocol layer (service provider). Typical functions are described in Table 3-2:

Table 3-2: Lower Layer Interface

Function	Description
Bind/Unbind	Registers and de-registers service user with service provider.
Connect/ Disconnect	Initiates peer-to-peer connection and disconnection procedures.
Data Transfer	Initiates peer-to-peer data transfer procedures.
Reset	Initiates peer-to-peer reset procedures.
Flow Control	Initiates flow control between service user and service provider.

The lower layer interface may be different for each product and may not exist for some products. This interface is described within the Service Definition of the appropriate product.

The lower interface of one product (service user) is designed to match the upper interface of the corresponding product (service provider).

3.1.3 Layer Management Interface

The layer management interface provides the management functions required to control and monitor the Trillium product. Typical functions are described in Table 3-3:

Table 3-3: Layer Management Interface

Function	Description
Configuration	Configures the protocol layer resources.
Statistics	Determines traffic loads and quality of service for the protocol layer.
Solicited Status	Indicates the current state of the protocol layer.
Unsolicited Status	Indicates a change in status of the protocol layer.
Accounting	Gathers accounting information from the protocol layer for billing purpose.
Tracing	Logs protocol messages to the layer manager for tracing purposes.
Control	Activates and deactivates protocol layer resources.

The layer management interface may be different for each product. This interface is described within the Service Definition of the appropriate product.

3.1.4 System Services Interface (SSI)

The System Services Interface provides the operating system functions required by the Trillium product. These functions are described in Table 3-4:

Table 3-4: System Services Interface

Function	Description
Initialization	Initializes the protocol layer.
Timer Management	Provides for the periodic activation of the protocol layer.
Task Scheduling	Registers, de-registers, activates, and terminates a task.
Memory Management	Allocates and deallocates variable sized buffers from memory pools.
Message Management	Initializes, adds, and removes data to and from messages.
Queue Management	Initializes, adds, and removes messages to and from queues.
Miscellaneous	Date and time management, error handling, and resource availability checking.

The System Services Interface is the same for each product. Not all System Services Interface functions may be used by each product. This interface is completely described within this document.

3.1.5 Service Access Points (SAPs)

TAPA follows the Open Systems Interconnection (OSI) reference model. At an interface between two layers in a protocol stack, there is a service user and a service provider. The service user accesses the services of the service provider at a service access point (SAP). Both the service user and the service provider have control structures to manage the interactions at the SAPs (for example, to store state of SAP and manage timers, and so on). There is a one-to-one mapping between a service user control block and a service provider control block, and this mapping defines the SAP, as shown in the following diagram:

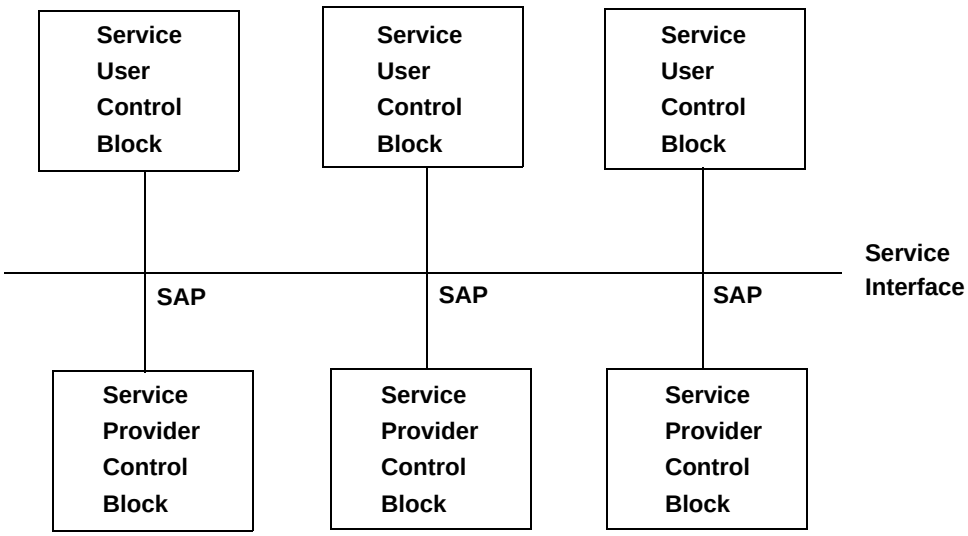


Figure 3-2: Service Access Points (SAPs)

The SAP control block is identified within the service provider with a **`spId`** (service provider SAP ID).

The SAP control block is identified within the service user with a **`suId`** (service user SAP ID).

3.1.6 Layer Coupling

As described earlier, layers or tasks communicate with each other either through direct function calls (tight coupling) or through message passing (loose coupling).

3.1.6.1 Tight Coupling

In tight coupling, interface primitives invoked from one task translate into direct function calls into the destination task.

Control returns to the calling task after the called task has completed processing the original primitive and any resultant primitives. In this sense, tight coupling provides a synchronous interface.

Since primitive invocations are nested, tight coupling may result in very deep (and indeterminate) function stacks. Because the calling layer might be called back by the destination layer, care must be taken to handle such events properly.

3.1.6.2 Loose Coupling

In loose coupling, interface primitives invoked from one task are packed into messages that are then sent to the destination task through the system services.

Control returns to the calling task immediately after it posts the message, before the destination task has seen or processed the primitive. In this sense, loose coupling provides an asynchronous interface. If the destination task needs to communicate something back to the source task, it does so through an interface primitive that gets sent in a message.

Messages posted between communicating tasks carry priority and routing information.

3.1.6.3 Interface Primitive Resolution

A primitive generated by a source layer must be routed to the destination layer. The information needed to route the primitive is provided in the **`Pst`** structure:

```
typedef struct pst          /* parameters for SPstTsk */
{
  ProcId    dstProcId;      /* destination processor ID */
  ProcId    srcProcId;      /* source processor ID */
  Ent       dstEnt;         /* destination entity */
  Inst      dstInst;        /* destination instance */
  Ent       srcEnt;         /* source entity */
  Inst      srcInst;        /* source instance */
  Prior     prior;          /* priority */
  Route     route;          /* route */
  Event     event;          /* event */
  Region    region;         /* region */
  Pool      pool;           /* pool */
  Selector  selector;       /* selector */
```

```
  U16 spare1;              /* spare for alignment */
}  Pst;
```

The sequence of operations for primitive resolution at a loosely coupled interface is depicted in the Figure 3-3 flow diagram:

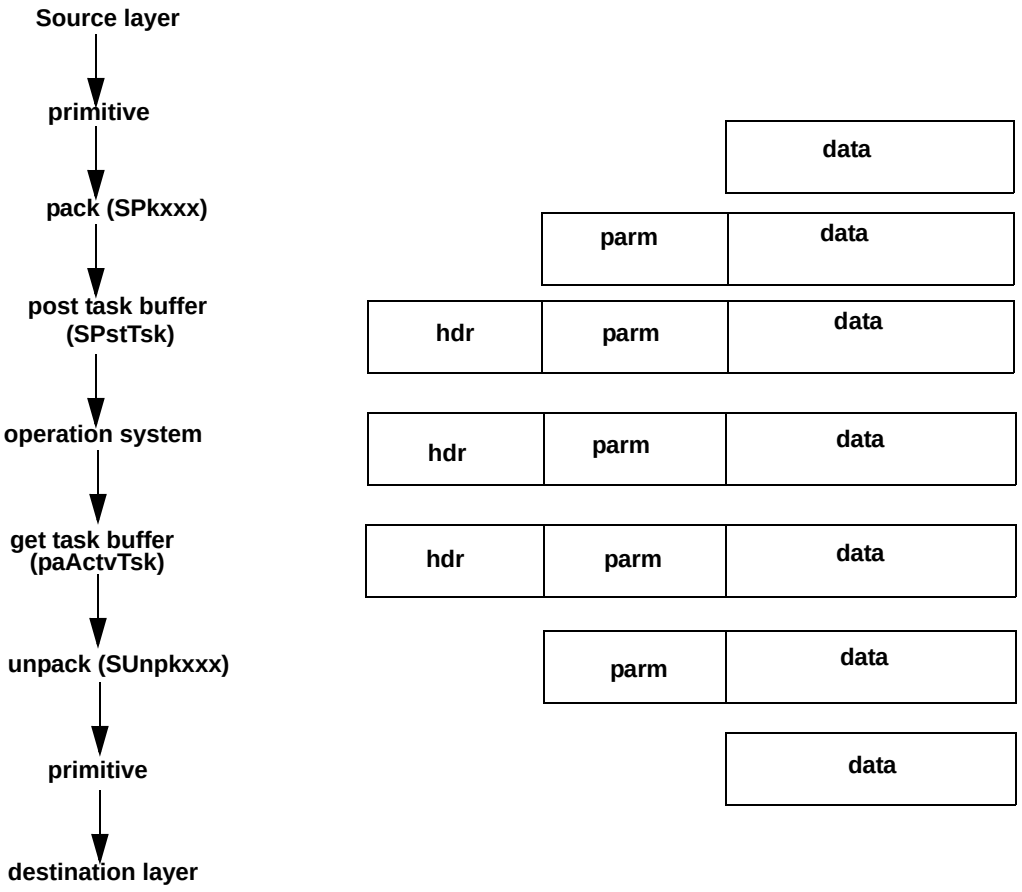


Figure 3-3: Operations for primitive resolution: loosely coupled interface

The following example for the **EcLiAmtBndReq** primitive shows how a primitive is resolved at a layer interface using the post structure (**SSINT2**). All primitives are called with the post structure as the first parameter. The primitive is resolved in the interface file by indexing into a table of function names using the selector field of the

```
post structure,       pst->selector.
PUBLIC S16 EcLiAmtBndReq(pst, suId, spId)
Pst    *pst;                /* post structure */
SuId   suId;                /* service user SAP ID */
SpId   spId;                /* service provider SAP ID */
{
    TRC3(EcLiAmtBndReq)

   /* jump to specific primitive depending on configured selector */
   RETVALUE((*ecLiAmtBndReqMt[pst->selector])(pst, suId, spId));
}
PRIVATE AmtBndReq ecLiAmtBndReqMt[EC_MAX_LIAMT_SEL] =
{
#ifdef LCECLIAMT
  cmPkAmtBndReq,           /* 0 - loosely coupled, Q.93B */
#else
  PtLiAmtBndReq,           /* 0 - loosely coupled, portable */
#endif
#ifdef AM
  AmUiAmtBndReq,           /* 1 - tightly coupled, Q.93B */
#else
  PtLiAmtBndReq,           /* 1 - tightly coupled, portable */
#endif
};
```

If the selector indicates tight coupling, the source layer primitive is translated into a direct function call in the destination layer.

If the selector indicates loose coupling, the source layer primitive is translated into a function that packs the primitive into a message. A message is allocated from **pst->region** and **pst->pool**. The primitive parameters are packed into this message. The primitive type is noted in the **pst->event field**. This message is then handed to system services, through the **SPstTsk** function, to get it to the destination, **pst->dstEnt, pst->dstInst and pst->dstProcId**, at the specified priority, **pst- >prior, and route, pst->route.**

```
PUBLIC S16 cmPkAmtBndReq(pst, suId, spId)
Pst    *pst;                /* post structure */
SuId   suId;                /* service user SAP ID */
SpId   spId;                /* service provider SAP ID */
{
  Buffer *mBuf; /* message buffer */
   TRC3(cmPkAmtBndReq)
  /* get a buffer for packing */
  SGetMsg(pst->region, pst->pool, &mBuf);
```

```
  /* pack parameters */
  CMCHKPKLOG(cmPkSpId, spId, mBuf, ECMATM001, pst);
  CMCHKPKLOG(cmPkSuId, suId, mBuf, ECMATM002, pst);

  /* post buffer */
  pst->event = (Event) AMT_EVTBNDREQ;
  RETVALUE(SPstTsk(pst, mBuf));
} /* end of cmPkAmtBndReq */
```

Once the system services gets the posted message, it queues it using a scheduling mechanism and eventually schedules the destination layer with the message, through the activation function, **amActvTsk**. The destination layer unpacks the original primitive from the message and invokes the desired function.

```
PUBLIC S16 amActvTsk(pst, mBuf)

Pst    *pst;               /* post structure */
Buffer *mBuf;              /* message buffer */
{
  TRC3(amActvTsk)
   /* call appropriate unpacking function, depending on event
      g1type */
  switch(pst->event)
  {
     /* upper layer primitives */
      case AMT_EVTBNDREQ: /* Upper Layer Bind Request */
        cmUnpkAmtBndReq(AmUiAmtBndReq, pst, mBuf);
        break;
     ...
      default:
#ifdef ERRCHK
        SError(EAM055, ERRZERO);
#endif
        RETVALUE(RFAILED);
  }
  SExitTsk();
   RETVALUE(ROK);
} /* end of amActvTsk */
```

3.1.7 Multiple Service Users and Service Providers

Each product supports multiple service users and/or service providers. This is accomplished by the product providing multiple upper service access points (SAPs) for its service users. Similarly, each product can access the services of its service provider through multiple lower SAPs.

The semantics of upper and lower SAPs, and whether any multiplexing is done between them, differs from product to product.

3.1.8 Single or Multi-Memory Architectures

As described earlier, the system can have multiple memory regions and pools.

Every SAP can be configured with a memory region and pool ID, so that all primitives going through that SAP use that memory pool. Different parts of the system may thus use different parts of the memory more effectively.

3.1.9 Single or Multitasking Architectures

Every SAP is configured with the identity of the destination task: A processor ID, an entity ID, and an instance ID, which is used by all primitives going through that SAP.

Thus, different tasks can belong to different processes and can reside on different processors.

3.1.10 Error Checking and Recovery

Extensive error checking and recovery mechanisms make the software robust enough to deal with normal error conditions as described below. Although error checks are important, it may sometimes be desirable to sometimes disable them for performance reasons; therefore, some level of user control is provided in selecting the granularity of error checking. This is done by the use of compile-time flags described below.

3.1.10.1 Types of Error Checks

Errors can be classified into certain categories, as follows.

3.1.10.1.1 Protocol Error Checks

These errors are related to the protocol layer that is being implemented. They are:

- Receiving an unexpected or improper protocol message.
- Timer expiry.

Such checks are always done, regardless of any error checking level selected by the user.

3.1.10.1.2 Interface Error Checks

These errors are related to function parameter and return status validation. They are:

- Illegal parameter values in interface primitives or common functions such as **cm_hash**.
- Failure status from SSPs of internal support functions.

There are two classes of error checks here, as follows.

3.1.10.1.3 Input Checks

These checks relate to validating the parameters supplied in an interface primitive or common function (for example, checking if the **spId** supplied in a **BndReq** primitive is for a configured SAP). Such checks may be disabled once the interfaces are debugged. They are done under compile flag **ERRCLS_INT_PAR**.

3.1.10.1.4 Output Checks

These checks relate to validating the return code from an interface primitive or common function (for example, checking if a call to **SGetSBuf** is successful). Output errors from system services and common functions are generally checked for.

3.1.10.1.5 Resource Errors

These relate to errors that result from scarcity of resources. They are:

- When a request to set up a new connection is received, there may not be memory to allocate a control point for this new connection.
- When encoding a protocol message or packing a primitive in a message, there may be no message buffers available.

Scarcity of resources is a "normal" condition, in the sense that it is very difficult to engineer a system to guarantee that resources are always available. However, there is a subtle distinction between the two types of resource error, as described below.

1. Get Resource Errors: Relate to situations in which a "new" resource is being requested and resource allocation must occur. For example, when a new message is being created, a new message buffer must be allocated using **SGetMsg**. Other examples from system services are:
   - **SGetSBuf**
   - **SGetMsg**
   - **SGetSMem**
   - **SCpyMsgMsg**

Such errors are always checked for, since they can occur at any time.

2. Add Resource Errors: Relate to situations in which a resource already exists, and a request is made to add more resources to it. For example, when more data needs to be added to an existing **mBuf** using **SaddPstMsg**. Other examples from system services are:
   - **SPkU8**
   - **SPkU16**
   - **SAddPreMsg**
   - **SAddPstMsg**

In this case, the system can be engineered so that these errors do not occur. For example, if the maximum size of a message is known beforehand, that can be allocated during the "get" resource stage, thus assuring that all future "add" requests succeeds. Such checks are done under compile flag **ERRCLS_ADD_RES**.

3.1.10.1.6 Debug Errors

These relate to errors that can happen only because of internal inconsistency within the layer. They are:

- An illegal state in the state machine.
- A default case for a switch statement.

Other examples from system services are:

- **SUnpkU8**
- **SUnpkU16**
- **SPrint**
- **SPrntMsg**

One can also view these as "impossible" conditions that does not happen when the code is bug-free. These errors can be disabled after the debugging phase of code development. Such checks are done under compile flag **ERRCLS_DEBUG**.

3.1.10.2 Granularity of Control Over Error Checks

The above discussion details certain categories of error checks that can be performed to make the layer more robust. However, it may not be desirable to perform all the error checks all the time. In other words, there must be the capability to disable some error checks, if there is sufficient assurance that the system is designed (and tested) to never encounter those errors. Certain error class flags (**ERRCLS_***) are defined that must surround the corresponding types of error checks, so they can be disabled at compile-time, if desired.

3.1.10.2.1 Error Class Flags

Table 3-5: Error Class Flags

Error Class Type	When Checked	Corresponding Flag
Protocol errors	Always (protocol messages, interface primitives)	(none)
Interface errors:		
• Input checks	• At black box boundaries (interface primitives, common functions)	• **ERRCLS_INT_PAR**
• Output checks	• At internal support functions	• **ERRCLS_DEBUG**
	• When meaningful (system services, common, and support functions)	• (none)
Resource errors:		
• Get resources	• Always	• (none)
• Add resources	• When resource failure is possible	• **ERRCLS_ADD_RES**
Debug errors	When debugging code (impossible conditions, conditions indicating bugs)	**ERRCLS_DEBUG**

Since multiple error classes can be enabled at the same time, the error class flags are defined as bit flags that can be combined into a composite **ERRCLASS** flag. The following #defines are defined in the **ssi.h** so they can be included in every layer-specific file:

```
#ifndef ERRCLS_ADD_RES
#define ERRCLS_ADD_RES 0x1
#endif /* ERRCLS_ADD_RES */

#ifndef ERRCLS_INT_PAR
#define ERRCLS_INT_PAR 0x2
#endif /* ERRCLS_INT_PAR */

#ifndef ERRCLS_DEBUG
#define ERRCLS_DEBUG   0x4
#endif /* ERRCLS_DEBUG */
```

```
/* Using ERRCLASS instead of ERRCHK to avoid any potential
   conflict with old common files */
#ifndef NO_ERRCLS
#define ERRCLASS (ERRCLS_ADD_RES|ERRCLS_INT_PAR|ERRCLS_DEBUG)
#else
#define ERRCLASS 0
#endif /* NO_ERRCLS */
```

Each error class can be overridden on the compiler command line (**DERRCLS_ADD_RES=0**). This can be done on a file-by-file basis, or on the **ENV** line in the **makefile** for global restrictions. The **-DNO_ERRCLS** flag disables all error checking regardless of the values of the **ERRCLS_*** defines.

3.1.10.3 Recovery Action Upon Error Detection

When an error is detected, the following actions are taken:

- Generate an error indication outside the layer boundaries, so it can be logged and perhaps acted upon.
- Return to a stable state within the product, and continue processing other events.

These actions are described as follows.

3.1.10.3.1 Generating Error Indications

Two methods are available:

1. Logging the error with system services.
2. Generating layer manager alarms.

These are described as follows.

3.1.10.3.2 Logging the Error With System Services

This is done through the system service primitive, **SLogError**, described in Section 4.14.8.8.

3.1.10.3.3 Generating Layer Manager Alarms

Layer manager alarms (status indications) are generated to inform the layer manager about the following:

- Error Conditions (for example, illegal interface primitive parameters).
- Protocol State Changes (for example, SAP (link) Up or Down).
- Warnings (for example, reaching resource usage thresholds).

In general, alarms are generated to report errors that indicate a system design or configuration problem. Normally, the layer manager is not able to recover from the error programmatically, but can report it to the "user", who can use this information to redesign or reconfigure the system.

3.1.10.3.4 Error Logging versus Alarms

Two mechanisms are available to generate error indications. The rule regarding which to use when an error is detected is simple:

- When an error is detected under an **`ERRCLS_*`** flag, it is logged through **`SLogError`**. Logging is thus required for interface input errors, add resource errors, and debug errors.
- When an error has significance to the system design and configuration, it is reported as an alarm. Interface input errors and resource errors, therefore, are reported as alarms.

Except for interface input errors, all other errors are either logged or reported as an alarm but not both (protocol errors are neither logged nor reported as alarms). Interface input errors are both logged and reported as alarms.

Each mechanism has its advantages:

- Logging is simple and does not require any system resources. It is always a tightly coupled, system service function call that may log to a file or terminal and return (or optionally halt). It is useful during code debugging because it points to the error quickly and efficiently.
- Alarms are layer-specific and can carry structured information (in the form of alarm structures), thus providing a greater level of detail for debugging. However, since the layer manager may be loosely coupled, sending an alarm requires system resources that may not be available. Alarms are useful in a production system where it is not feasible to halt and debug the source.

3.1.10.3.5 Debug Printing

Debug printing is often included in the layer code to follow the information flow through the layer to get a better trace for debugging problems. However, debug printing is not an alternative to error logging or alarms. Consider these key points:

- If it is not an error condition, there is no logging or alarms. Instead, if it is useful to print something at this point, debug printing is used; that is, once a primitive is received, its parameter values can be printed.
- If it is an error condition, logging or alarms are used as described above. Debug printing is not used, because logging fulfills that purpose.

3.1.10.3.6 Returning to Stable State

In addition to generating error indications, the layer must also return to stable state after an error is encountered. This action is protocol- and event-specific. Some possibilities are described in the next sections.

3.1.10.3.7 Ignoring the Error

In the simplest case, the layer can ignore the error event and continue to process the event. Such cases are relatively rare. An example is when the layer wants to get a **`timestamp`** for an alarm and **`SGetDateTime`** fails, it can ignore it and continue, since a **`timestamp`** is not critical to the operation.

3.1.10.3.8 Truncating the Event Process

The layer can truncate the processing of a current event after an error occurs. That is, when the layer needs to send out a PDU and it cannot acquire a message buffer to do that, it can still assume that the PDU was indeed sent out and then lost in the network, that is, if the protocol can deal with unreliable PDU transmission.

3.1.10.3.9 Rolling Back the State

In this case, the layer attempts to roll back the state to some stable state prior to the error condition. When configuring a SAP, if a series of static buffers needs to be allocated and one of them fails, the previously allocated buffers must be returned to the pool.

4

System Services

This section describes all functions defined at the System Services Interface. Not all functions defined here are used by all products. The specific functions required by each product are specified in the Service Definition for that product.

4.1 Initialization Functions

Table 4-1: Initialization Functions

Name	Description
SRegInit	Register activation function - initialization.
xxActvInit	Initialization function for a task.

4.2 Timer Functions

Table 4-2: Timer Functions

Name	Description
SRegTmr	Register activation function - timer.
SDeRegTmr	De-register activation function - timer.
xxActvTmr	Timer activation function for a task.

4.3 Event Function

Table 4-3: Event Functions

Name	Description
SRegActvTsk	Register message handling function.
SDeregInitTskTmr	De-register initialization, timer, and message handling activation functions.
SPstTsk	Post a message to a task.
xxActvTask	Message handling function for a task.
SExitTsk	Exit a function activation.

4.4 Driver Functions

Table 4-4: Driver Functions

Name	Description
SRegDrvrTsk	Register driver activation function.
SAlignDBufEven	Align dynamic buffer data on an even boundary.
SChkMsg	Check message for certain hardware requirements prior to transmission.
SSetIntPend	Notify system services of a pending interrupt.
SEnbInt	Enable interrupt.
SDisInt	Disable interrupt.
SHoldInt	Hold interrupt.
SRelInt	Release interrupt.
SGetVect	Get vector.
SPutVect	Put vector.
SGetEntInst	Get current entity ID and instance ID.
SSetEntInst	Set current entity ID and instance ID.
SGetDBuf	Allocate a dynamic buffer (from a dynamic pool).
SPutDBuf	Deallocate a dynamic buffer (from a dynamic pool).
SAddDBufPst	Append dynamic buffer to message buffer.
SAddDBufPre	Prepend dynamic buffer to message buffer.

Name	Description
SRemDBufPst	Remove dynamic buffer from end of message buffer.
SRemDBufPre	Remove dynamic buffer from start of message buffer.
SGetDataRx	Get data start pointer and size for receive (empty) buffer.
SGetDataTx	Get data start pointer and size for receive (full) buffer.
SInitNxtDBuf	Initialize next dynamic buffer in message buffer.
SGetNxtDBuf	Get next dynamic buffer.
SChkNxtDBuf	Check for the existence of a next dynamic buffer in a message buffer.
SUpdMsg	Update message buffer (by appending) with dynamic buffer.

4.5 Memory Management Functions

The memory management functions allocate and deallocate pools and buffers, static or dynamic. The following functions are used for memory management:

Table 4-5: Memory Management Functions

Name	Description
SGetSMem	Allocate a pool of static memory.
SPutSMem	Deallocate a pool of static memory.
SGetSBuf	Allocate a static buffer (from a static pool).
SPutSBuf	Deallocate a static buffer (into a static pool).

4.6 Message Functions

The message management functions initialize, add, and remove data to and from messages utilizing dynamic buffers. The following functions are used for message management:

Table 4-6: Message Functions

Name	Description
SGetMsg	Allocate a message (from a dynamic pool).
SPutMsg	Deallocate a message (into a dynamic pool).
SInitMsg	Initialize a message.
SFndLenMsg	Find the length of a message.
SExamMsg	Examine an octet at a specified index in a message.
SRepMsg	Replace an octet at a specified index in a message.
SAddPreMsg	Add an octet to the beginning of a message.
SAddPstMsg	Add an octet to the end of a message.
SRemPreMsg	Remove an octet from the beginning of a message.
SRemPstMsg	Remove an octet from the end of a message.
SAddPreMsgMul	Add multiple octets to the beginning of a message.
SAddPstMsgMult	Add multiple octets to the end of a message.
SGetPstMsgMult	Add multiple zero octets to the end of a message.
SRemPreMsgMult	Remove multiple octets from the beginning of a message.
SRemPstMsgMult	Remove multiple octets from the end of a message.
SCpyFixMsg	Add multiple octets to a message at a specified index.
SCpyMsgFix	Copy octets from a message at a specified index into fixed (contiguous) storage.
SCpyMsgMsg	Copy a message into a newly allocated message.
SCatMsg	Concatenate two messages.
SSegMsg	Segment a message into two.
SCompressMsg	Compress the data storage for a message.
SAddMsgRef	Increment the message reference count.
SPkS8	Add a signed 8 bit value to a message.
SPkU8	Add an unsigned 8 bit value to a message.

Table 4-6: Message Functions

Name	Description
SPkS16	Add a signed 16 bit value to a message.
SPkU16	Add an unsigned 16 bit value to a message.
SPkS32	Add a signed 32 bit value to a message.
SPkU32	Add an unsigned 32 bit value to a message.
SUnpkS8	Remove a signed 8 bit value from a message.
SUnpkU8	Remove an unsigned 8 bit value from a message.
SUnpkS16	Remove a signed 16 bit value from a message.
SUnpkU16	Remove an unsigned 16 bit value from a message.
SUnpkS32	Remove a signed 32 bit value from a message.
SUnpkU32	Remove an unsigned 32 bit value from a message.

4.7 Queue Management Functions

The queue management functions initialize, add, and remove messages to and from queues. The following functions are used for queue management:

Table 4-7: Queue Management Functions

Name	Description
SInitQueue	Initialize a queue.
SQueueFirst	Add a message to the beginning of a queue.
SQueueLast	Add a message to the end of a queue.
SDequeueFirst	Remove a message from the beginning of a queue.
SDequeueLast	Remove a message from the end of a queue.
SFlushQueue	Remove all messages from a queue and deallocate them.
SCatQueue	Concatenate two queues.
SFndLenQueue	Find the number of messages in a queue.
SAddQueue	Add to a queue.
SRemQueue	Remove from a queue.
SExamQueue	Examine a queue.

4.8 Miscellaneous Functions

The miscellaneous functions are used for date and time management, error handling, and resource availability checking. The following miscellaneous functions are used:

Table 4-8: Miscellaneous Functions

Name	Description
SFndProcId	Find processor ID on which a task is running.
SSetProcId	Set the local processor ID.
SSetDateTime	Set real date and time.
SGetDateTime	Get real date and time.
SGetSysTime	Get system time.
SError	Handle an error (OBSOLETE).
SLogError	Handle an error.
SChkRes	Check free memory in a pool.
SRandom	Generate a random number.
SPrint	Print a preformatted string to the default display device.
SDisplay	Display a preformatted string on a specified display device.
SPrntMsg	Print a message's data part.

4.9 Multi-Threaded Primitives

The following primitives are now part of the System Services Interface. These primitives are available only when the special compiler flag -DMT is enabled. To date, the only environment in which Trillium supports these interfaces is Solaris 2.x. In the future, more environments are available to support these interface primitives. Currently, no portable protocol layers use these primitives. However, some of our Solaris-specific products do leverage these primitives internally.

Table 4-9: Multi-Threaded Primitives

Name	Description
SGetMutex	Get (allocate) a mutex.
SPutMutex	Put (free) a mutex.
SLockMutex	Lock a mutex.
SUnlockMutex	Unlock a mutex.
SGetCond	Get (allocate) a condition variable.
SPutCond	Put (free) a condition variable.
SCondWait	Wait on a condition variable
SCondSignal	Generate a signal on a condition variable.
SCondBroadcast	Broadcast a signal on a condition variable.
SGetThread	Create a thread of execution.
SPutThread	Destroy a thread of execution.
SThreadYield	Yield to another thread.
SThreadExit	Cause thread to exit.
SSetThreadPrior	Set a thread's priority.
SGetThreadPrior	Get a thread's priority.

4.10 Microsoft Windows NT Kernel Primitives

The following primitives are available only in Trillium's Microsoft Windows NT Kernel environment:

Table 4-10: Kernel Primitives

Name	Description
SPutIsrDpr	Put ISR and DPR (interrupt service and deferred procedure call).
SSyncInt	Synchronize interrupt.

4.11 Multi-core Support Functions

The following functions are part of SSI with multi-core enhancements. The functions are defined in **SS_Task.c**:

Table 4-11: Multi-core Support Functions

Name	Description
SRegCpuInfo	This function registers the total number of cores and the number of threads per core, and the number of available threads for SSI use.
SSetAffinity	This function is used to statically set the affinity of system tasks (threads) to CPU cores.
SGetAffinity	This function is used for knowing the current affinity status of a system task (thread).

4.12 Memory Management, Debugging, and Fault Detection Support Functions

The following functions are part of SSI with memory management, debugging, and fault detection enhancements. The functions are defined in **`SS_Task.c`**:

Table 4-12: Memory Management, Debugging and Fault-detection Support Functions

Name	Description
`SGetThrdProf`	This function is used to get the thread profiling information.
`SRegCfgWd`	This function is used to configure the watchdog functionality on the node.
`SDeregCfgWd`	This function is used to unconfigure the watchdog functionality.
`SStartHrtBt`	This function is used to initiate the heartbeat mechanism on the node.
`SStopHrtBt`	This function is used to stop the heartbeat mechanism on the node.
`SRegLogCfg`	This function is used to configure/register the SSI logger.
`SDeregLogCfg`	This function is used to unconfigure/un-register the SSI logger.
`SHstGrmInfoShow`	This function is used to display the memory histogram information.
`SLogLkInfo`	This function is used to display the memory leak information if any.
`SRegTskInfo`	This function is used to register TAPA tasks based on a configuration file.

4.13 Type Definitions

Following are the common type definitions that recur in the description of the specific system services interface functions. They can be found in **ssi.x**:

```
typedef S16 Status;          /* status */
typedef U32 Ticks;           /* system clock ticks */
typedef S16 MsgLen;          /* message length */
typedef S16 BufQLen;         /* buffer queue length */
typedef S16 Order;           /* message or queue order */
#ifdef DOS
typedef U16 Size;            /* size */
#else
typedef U32 Size;            /* size */
typedef S32 PtrOff;          /* signed pointer offset */
#endif
typedef U32 Qlen;            /* queue length */
typedef S16 RegSize;         /* region size */
typedef S16 DpoolSize;       /* dynamic pool size */
typedef U16 Random;          /* random number */
typedef S16 Seq;             /* sequence */
typedef U32 ErrCls;          /* Error Class */
typedef U32 ErrCode;         /* Error Code  */
typedef U32 ErrVal;          /* Error Value */
typedef S16 VectNmb;         /* vector number */
typedef S16 Ttype;           /* task type */
typedef struct dateTime      /* date and time */
{
  U8 month;                  /* month */
  U8 day;                    /* day */
  U8 year;                   /* year */
  U8 hour;                   /* hour - 24 hour clock */
  U8 min;                    /* minute */
  U8 sec;                    /* second */
  U8 tenths;                 /* tenths of second */
} DateTime;
typedef struct duration      /* duration */
{
  U8 days;                   /* days */
  U8 hours;                  /* hours */
  U8 mins;                   /* minutes */
  U8 secs;                   /* seconds */
  U8 tenths;                 /* tenths of seconds */
} Duration;
typedef struct mem           /* memory */
{
  Region region;             /* region */
  Pool   pool;               /* pool */
  U16    spare;              /* spare for alignment */
} Mem;
```

```
typedef Mem MemoryId;       /* memory ID */
```

The following **typedef**s are available only when running in a multi-threaded environment (-DMT):

```
typedef S32 MtCondId;
typedef S32 MtMtxId;
typedef S32 MtThrdId;
typedef S32 MtThrdFlags;
typedef S32 MtThrdPrior;
typedef Void *(MtThrd) ARGS((Void *));
```

The following typedefs are defined in **gen.x**.:

```
typedef S16 Elmnt;          /* element */
typedef S16 ElmntInst1;     /* element instance 1 */
typedef S16 ElmntInst2;     /* element instance 2 */
typedef S16 ElmntInst3;     /* element instance 3 */
typedef struct elmntId      /* element ID */
{
  Elmnt      elmnt;         /* element */
  ElmntInst1 elmntInst1;    /* element instance 1 */
  ElmntInst2 elmntInst2;    /* element instance 2 */
  ElmntInst3 elmntInst3;    /* element instance 3 */
} ElmntId;
```

The following typedefs are defined in **envdep.h**:

```
/* pointer to initialization function returning S16 */
typedef S16  (*PAIFS16) ARGS((Ent ent, Inst inst, Region region,
                              Reason reason));

/* pointer to timer activation function returning S16 */
typedef S16  (*PFS16) ARGS((void));

/* pointer to message activation function returning S16 */
typedef S16 (*ActvTsk) ARGS((Pst *pst, Buffer *mBuf);
```

4.14 Function Specifics

The following section describes the specific calling parameters and sequence for the System Services Interface functions. Each service interface function is described in terms of the following:

Header

Primitive name.

Name

Description of primitive name.

Direction

Calling direction of primitive.

Supplied

Whether the primitive is supplied as part of the standard product deliverable.

Synopsis

K&R declaration of function.

Parameter

Description of parameters and their allowable values.

Description

Description of primitive usage.

Returns

Description of return value and it's allowable value.

4.14.1 Initialization Functions

4.14.1.1 SRegInit

Name

Register initialization function for a task.

Direction

Layer Manager to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRegInit(ent, inst, initFnct)
Ent      ent;
Inst     inst;
PAIFS16  initFnct;
```

Parameters

`ent`

Entity ID of task to initialize. Allowable values: 0 - 255.

`inst`

Instance of task to initialize. Allowable values: 0 - 255.

`initFnct`

Pointer to initialization function in the task, typically, **`xxActvInit`**.

Description

This function is used to register an initialization function for a task. The system services invokes the function passed to it once before scheduling the task with any other events. Typically, it schedules the function immediately. The initialization function is used by the task to initialize its global variables.

The data flow is:

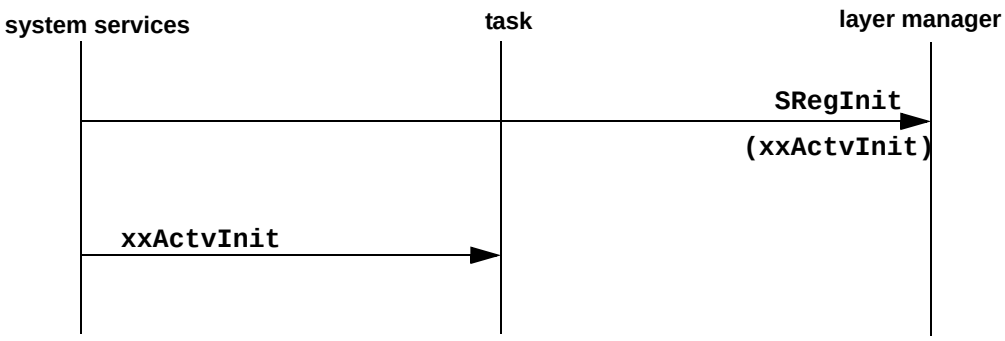


Figure 4-1: Data flow: initialization function

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.1.2 xxActvInit

Name

Initialization function for a task. Typically, this function is named **xxActvInit**, where **xx** is the two-letter prefix for the product represented by the task.

Direction

System Services to Layer Software.

Supplied

- Protocol layer product: Yes.
- System services product: No.

Synopsis

```
PUBLIC S16 xxActvInit (ent, inst, region, reason)
Ent      ent;
Inst     inst;
Region   region;
Reason   reason;
```

Parameters

`ent`

Entity ID of task to initialize. Allowable values: 0 - 255.

`inst`

Instance ID of task to initialize. Allowable values: 0 - 255.

`region`

Region ID of memory from which the task may allocate a static pool for its data structures. Allowable values: 0 - 255. The meaning of the value is determined by the system architecture.

`reason`

Reason for initialization. Allowable values:

```
#define NRM_TERM     0     /* normal termination */
#define PWR_UP       1     /* power up */
#define SWITCH       2     /* switch depressed */
#define SW_ERROR     3     /* software error */
#define DMT_FIRED    4     /* deadman timer fired */
#define EXTERNAL     5     /* external, another board */
#define SHUTDOWN     6     /* shutdown interrupt */
```

The protocol layers do not currently make use of this parameter.

Description

The system services invokes this function, passed to it through **`SRegInit`**, exactly once, before scheduling the task with any other events. The task uses the initialization function to initialize its global variables.

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.2 Timer Functions

4.14.2.1 SRegTmr

Name

Register timer activation function for a task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRegTmr (ent, inst, period, tmrFnct)
Ent      ent;
Inst     inst;
S16      period;
PFS16    tmrFnct;
```

Parameters

ent

Entity ID of task registering the timer. Allowable values: 0 - 255.

inst

Instance of task registering the timer. Allowable values: 0 - 255.

period

Period, in system ticks, between system services' successive scheduling of the timer function (**tmrFnct**) in the task. The value represents the physical (real) time resolution of timer activations required by the task.

tmrFnct

Timer function, typically **xxActvTmr**.

Description

This function is used by a task to register a timer function. The system services periodically invokes the function passed to it at the specified intervals. The timer function is used by the task to manage its internal timers (for example, protocol timers).

A **0.1** second timer tick resolution is recommended, although there are no system dependencies on this value.

Typically, one timer function is registered by a task for each different timer resolution required by it; however, some layers register more than one timer.

The data flow is:

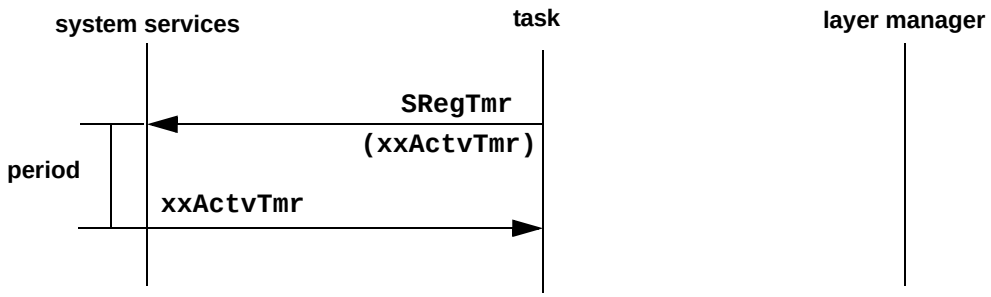


Figure 4-2: Data flow: timer function

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.2.2 SDeregTmr

Name

De-register timer activation function for a task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SDeregTmr (ent, inst, period, tmrFnct)
Ent      ent;
Inst     inst;
S16      period;
PFS16    tmrFnct;
```

Parameters

ent

Entity ID of task de-registering the timer. Allowable values: 0 - 255.

inst

Instance of task de-registering the timer. Allowable values: 0 - 255.

period

Period, in system ticks, between system services' successive scheduling of the timer function (**tmrFnct**) in the task. The value represents the physical (real) time resolution of timer activations required by the task.

tmrFnct

Timer function, typically **xxActvTmr**.

Description

This function is used by a task to de-register a timer function. The timer resource is deleted from the system.

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.2.3 xxActvTmr

Name

Timer activation function for a task. Typically, this function is named **xxActvTmr**, where **xx** is the two-letter prefix for the product represented by the task.

Direction

System Services to Layer Software.

Supplied

- Protocol layer product: Yes.
- System services product: No.

Synopsis

```
PUBLIC S16 xxActvTmr()
```

Parameters

None.

Description

This function is scheduled periodically by the system services, as registered with **`SRegTmr`**. The timer function is used by the task to manage its internal timers (for example, protocol timers).

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.3 Event Functions

4.14.3.1 SRegActvTsk

Name

Register message activation function for a task.

Direction

Layer Manager to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRegActvTsk (ent, inst, ttype, prior, actvTsk)
Ent      ent;
Inst     inst;
Ttype    ttype;
Prior    prior;
ActvTsk  actvTsk;
```

Parameters

ent

Entity ID of task to activate. Allowable values: 0 - 255.

inst

Instance of task to activate. Allowable values: 0 - 255.

ttype

Task type. Allowable values:

```
#define TTNORM   0x01     /* normal task - non preemptive */
#define TTPERM   0x02     /* permanent task */
```

```
prior
/* Priority of task. Allowable values: */
#define PRIOR0    0x00      /* priority 0 - highest */
#define PRIOR1    0x01      /* priority 1 */
#define PRIOR2    0x02      /* priority 2 */
#define PRIOR3    0x03      /* priority 3 - lowest */
```

actvTsk

Activation function, typically **xxActvTsk**

Description

This function is used to register a message activation function for a task. The activation function has an assigned priority that may be used to receive messages of any priority less than or equal to the assigned priority.

If the task type is **TTNORM** or **TTPREEMPT**, the system services schedules the activation function passed to it whenever a message is received that is destined for this task. The task uses the activation function to receive and handle messages from other tasks.

If the task type is **TTPERM**, the system services schedules the activation function passed to it at irregular intervals. Typically, it is scheduled whenever there are no pending messages.

The data flow is:

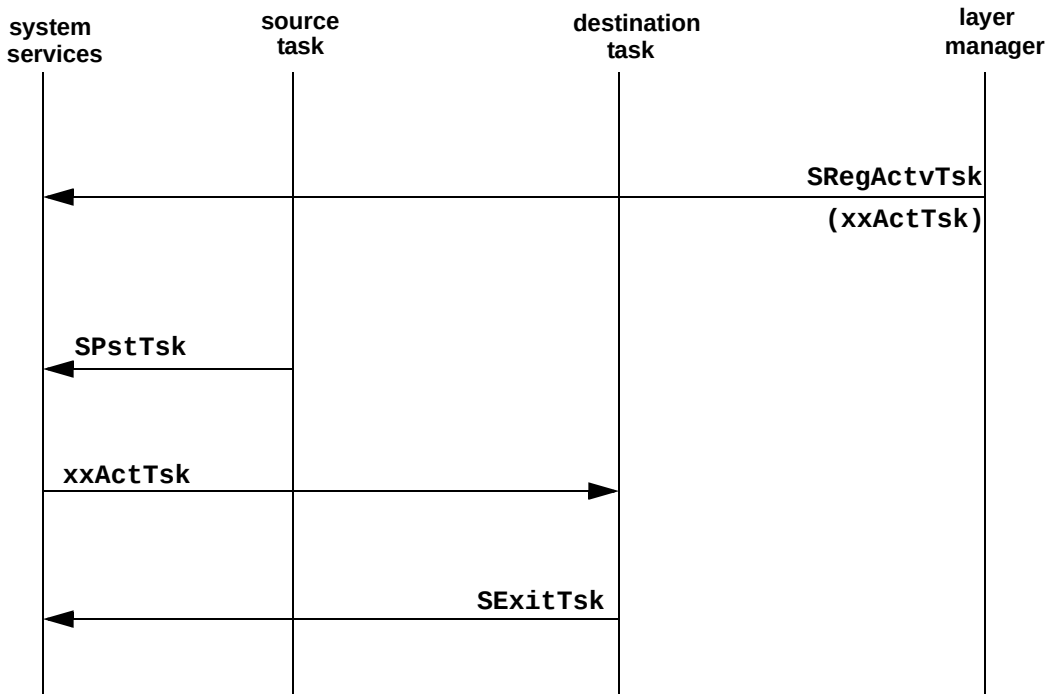


Figure 4-3: Data flow: event function

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.3.2 SDeregInitTskTmr

Name

De-register initialize, message, and timer activation functions for a task.

Direction

Layer Manager to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SDeregInitTskTmr (ent, inst)
Ent      ent;
Inst     inst;
```

Parameters

ent

Entity ID of task to deactivate. Allowable values: 0 - 255.

inst

Instance ID of task to deactivate. Allowable values: 0 - 255.

Description

This function is used to de-register the initialization, message, and timer activation functions for a task.

Returns

ROK OK.

RFAILED Failed: Illegal parameters, unable to free memory.

4.14.3.3 SPstTsk

Name

Post a message to a task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPstTsk (pst, mBuf)
Pst      *pst;
Buffer   *mBuf;
```

Parameters

pst

Pointer to post structure. Previously described. It contains information to identify the source task, destination task, message priority, and so on.

mBuf

Task buffer.

Description

This function takes a message from a source task and routes it to a destination task at the specified priority.

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.3.4 xxActvTsk

Name

Message activation function for a task.

Direction

System Services to Layer Software.

Supplied

- Protocol layer product: Yes.
- System services product: No.

Synopsis

```
PUBLIC S16 xxActvTsk (pst, mBuf)
Pst      *pst;
Buffer   *mBuf;
```

Parameters

pst

Pointer to post structure. Previously described. It contains information to identify the source task, destination task, and message priority, and so on.

mBuf

Message to be received by the destination task.

Description

This function is used by the system services to activate a task whenever a message is received for that task. This function must be registered earlier using **SRegActvTsk**. It must exit with a call to **SExitTsk** under the preemptive scheduler.

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.3.5 SExitTsk

Name

Exit an activation function in a task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SExitTsk()
```

Parameters

None.

Description

This function exits from a message activation function in a task. All such functions (that is, the **`xxActvTsk`** and **`xxActvTm`**r functions) must exit with a call to this function. This routine allows "task conscious" schedulers to monitor task activations.

Returns

ROK OK.

4.14.4 Driver-Related Scheduling Functions

The functions in this section are provided for use by two types of tasks: Driver Tasks (tasks designated for moving messages between two or more arbitrary boundaries), and Physical Layer Tasks (tasks that interface with I/O hardware).

4.14.4.1 SRegDrvrTsk

Name

Register driver activation function for drivers that route system service messages off board.

Direction

Driver Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRegDrvrTsk (inst, low, high, actvTsk, isTsk)
Inst    inst;
ProcId  low;
ProcId  high;
ActvTsk actvTsk;
ISTsk   isTsk;
```

Parameters

`inst`

Instance ID of the driver routine. Passed back upon activation.

`low`

Low boundary of processor ID(s) messages that this driver instance can deliver.

`high`

High boundary of processor ID(s) messages that this driver instance can deliver.

`actvTsk`

Activation function that system services must call when a message is received for a processor ID within the range of low and high (inclusive).

isTsk

Interrupt service routine that system services must call when its interrupt pending flag is set through a call to **SSetIntPend()**.

Description

This function registers a driver routine, which is an entity capable of routing messages to foreign processor ID(s). All messages destined for processor ID(s) within the registered range must be passed to the activation function for delivery. Upon receipt of an interrupt, the driver entity must notify system services of a pending interrupt through the **SSetIntPend()** system service routine. As soon as possible (asynchronously), system services must then schedule the is **Tsk** to process the interrupt.

Returns

ROK OK.

RFAILED Failed: Illegal parameters.

4.14.4.2 SAlignDBufEven

Name

Align data portion.

Direction

Driver Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAlignDBufEven(dBuf)
Buffer *dBuf;
```

Parameters

dBuf

Data buffer.

Description

This function aligns the data portion of a data buffer on an even byte boundary. This routine is required for certain hardware architectures that require data to be aligned on even boundaries.

Returns

ROK OK.

RFAILED Failed.

4.14.4.3 SChkMsg

Name

Check message for certain hardware requirements.

Direction

Driver Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SChkMsg(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Message buffer pointer.

Description

This function checks that the first data buffer in a message contains at least two bytes. This routine is required by 68302/68360 processors to insure accurate FISU generation for SS7.

Returns

ROK OK.

RFAILED Failed.

4.14.4.4 SSetIntPend

Name

Notify system services of a pending interrupt.

Direction

Driver Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSetIntPend (id, flag)
U16  id;
Bool flag;
```

Parameters

`id`

Channel ID: Instance ID of channel that detected or processed interrupt.

`flag`

`TRUE` or **`FALSE`**: Set interrupt pending flag or clear interrupt pending flag.

Description

This function sets the Interrupt pending flag to **`TRUE`** or **`FALSE`** based on the value of the flag passed to it.

Upon receipt of interrupt, the driver entity notifies system services of a pending interrupt through this system service routine. As soon as possible (asynchronously), system services must then schedule the appropriate registered interrupt service task to process the interrupt.

Returns

ROK OK.

RFAILED Failed.

4.14.4.5 SExitInt

Name

Exit Interrupt.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SExitInt()
```

Parameters

None.

Description

This function exits from an interrupt. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.6 SEnbInt

Name

Enable Interrupt.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SEnbInt()
```

Parameters

None.

Description

This function enables interrupts. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.7 SDisInt

Name

Disable Interrupts.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SDisInt()
```

Parameters

None.

Description

This function disables interrupts. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.8 SHoldInt

Name

Hold Interrupt.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SHoldInt()
```

Parameters

None.

Description

This function prohibits interrupts from being enabled until release interrupt. It must be called when interrupts are disabled, and prior to any call to system services, either by entry to an interrupt service routine or by explicit call to disable interrupt.

This function may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.9 SRelInt

Name

Release Interrupt.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SRelInt()
```

Parameters

None.

Description

This function allow interrupts to be enabled. It may be called by the operating system or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.10 SGetVect

Name

Get Vector.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetVect (vectNmb, vectFnct)
VectNmb  vectNmb;
PIF      *vectFnct;
```

Parameters

vectNmb

Vector number.

vectFnct

Vector function.

Description

This function gets the function address stored at the specified interrupt vector. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.11 SPutVect

Name

Put Vector.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutVect (vectNmb, vectFnct)
VectNmb  vectNmb;
PIF      vectFnct;
```

Parameters

vectNmb

Vector number.

vectFnct

Vector function.

Description

This function puts the function address at the specified interrupt vector. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.12 SGetEntInst

Name

Get entity and instance ID.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetEntInst(ent, inst)
Ent   *ent;
Inst  *inst;
```

Parameters

ent

Pointer to location where the entity ID is placed.

inst

Pointer to location where instance ID is placed.

Description

This function gets the current entity and instance. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.13 SSetEntInst

Name

Set entity and instance ID.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSetEntInst(ent, inst)
Ent   ent;
Inst  inst;
```

Parameters

ent

Entity ID. Allowable values: 0 - 255.

inst

Instance ID. Allowable values: 0 - 255.

Description

This function sets the current entity and instance. It may be called by the OS or layer1 hardware drivers.

Returns

ROK OK.

RFAILED Failed.

4.14.4.14 SGetDBuf

Name

Allocate dynamic buffer.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SGetDBuf(region, pool, dBuf)
Region region;
Pool   pool;
Buffer **dBuf;
```

Parameters

region

Region from which to allocate buffer.

pool

Pool from which to allocate buffer.

dBuf

Returned value of dynamic buffer.

Description

This function allocates a dynamic (data payload) buffer for use in constructing messages.

Few portable layers call this function directly. Generally, driver layers and physical layers use this call for I/O.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.4.15 SPutDBuf

Name

Deallocate a dynamic buffer.

Direction

Layer Software to System Services.

Supplied

- Driver layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SPutDBuf (region, pool, dBuf)
Region region;
Pool   pool;
Buffer *dBuf;
```

Parameters

region

Region from which to allocate buffer.

pool

Pool from which to allocate buffer.

dBuf

Pointer to dynamic buffer.

Description

This function deallocates a dynamic (data payload) buffer which is put back to the specified dynamic memory pool.

Few portable layers call this function directly. Generally, driver layers and physical layers use this call for I/O.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.16 SAddDBufPst

Name

Add a data buffer to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SAddDBufPst (mBuf, dBuf)
Buffer *mBuf;
Buffer *dBuf;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Pointer to data buffer.

Description

This function adds a data buffer to the end of the specified message buffer. If the message is empty, the data buffer is placed in the message. If the message is not empty, the data buffer is added at the end of the message.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.17 SAddDBufPre

Name

Add a data buffer to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SAddDBufPre (mBuf, dBuf)
Buffer *mBuf;
Buffer *dBuf;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Pointer to data buffer.

Description

This function adds a data buffer to the front of the specified message buffer. If the message is empty, the data buffer is placed in the message. If the message is not empty, the data buffer is added at the front of the message.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.18 SRemDBufPst

Name

Remove a data buffer from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SRemDBufPst (mBuf, dBuf)
Buffer *mBuf;
Buffer **dBuf;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Pointer to data buffer.

Description

This function removes a data buffer from the end of the specified message buffer. If the message is empty, the pointer to the data buffer is set to null, and the return is OK: Data not available. If the message is not empty, the pointer to the buffer is set to the last data buffer in the message, the last data buffer is removed from the message, and the return is OK.

Returns

ROK OK.

ROKDNA OK: Data not available.

RFAILED Failed: Error.

4.14.4.19 SRemDBufPre

Name

Remove a data buffer from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SRemDBufPre (mBuf, dBuf)
Buffer *mBuf;
Buffer **dBuf;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Pointer to data buffer.

Description

This function removes a data buffer from the front of the specified message buffer. If the message is empty, the pointer to the data buffer is set to null, and the return is OK: Data not available. If the message is not empty, the pointer to the buffer is set to the first data buffer in the message, the first data buffer is removed from the message, and the return is OK.

Returns

ROK OK.

ROKDNA OK: Data not available.

RFAILED Failed: Error.

4.14.4.20 SGetDataRx

Name

Get data start pointer and size for buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SGetDataRx (dBuf, pad, retDatPtr, retDatLen)
Buffer *dBuf;
MsgLen  pad;
Data   **retDatPtr;
MsgLen *retDatLen;
```

Parameters

dBuf

Pointer to data buffer.

pad

Pad within the data buffer.

retDatPtr

Return data pointer.

retDatLen

Return data length.

Description

This function returns a data pointer to the data payload, and the length of the payload in a receive (empty) data buffer. The size here is the maximum number of data bytes that can be placed in the data buffer.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.21 SGetDataTx

Name

Get data start pointer and size for buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SGetDataTx (dBuf, retDatPtr, retDatLen)
Buffer *dBuf;
Data   **retDatPtr;
MsgLen *retDatLen;
```

Parameters

dBuf

Pointer to data buffer.

retDatPtr

Return data pointer.

retDatLen

Return data length.

Description

This function returns a data pointer to the data payload, and the length of the payload in a transmit (filled) data buffer. The size here is the actual number of data bytes in the data buffer.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.22 InitNxtDBuf

Name

Initialize next data buffer in message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SInitNxtDBuf(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Pointer to message buffer.

Description

This function initializes the next data buffer ID in the message to point to the first data buffer in the message. It must be called prior to calling **SgetNxtDBuf**. This primitive is used when special driver routines need to traverse chained (or non-chained) buffer structures, usually for transmission of raw data across physical interfaces.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.4.23 SGetNxtDBuf

Name

Get next data buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SGetNxtDBuf(mBuf, dBuf)
Buffer *mBuf;
Buffer **dBuf;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Data buffer to be returned.

Description

This function returns the next data buffer in a message. **SInitNxtDBuf** must be called prior to calling this system service routine.

Returns

ROK OK.

ROKDNA OK: Data not available.

RFAILED Failed: Error.

4.14.4.24 SChkNxtDBuf

Name

Check for a data buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SChkNxtDBuf(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Pointer to message buffer.

Description

This function checks for the existence of a next data buffer in a message. If the message has no data buffer, it returns OK: Data not available.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.4.25 SUpdMsg

Name

Append a data buffer to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SUpdMsg (mBuf, dBuf, dLen)
Buffer *mBuf;
Buffer *dBuf;
MsgLen dLen;
```

Parameters

mBuf

Pointer to message buffer.

dBuf

Data buffer to be appended.

dLen

Number of bytes in data buffer.

Description

This function updates a message by adding a data buffer to it. The data buffer is always appended to the message. Additionally, the message length is updated with **dLen**. This primitive differs from **SAddDBufPst** in the fact that it updates the message length.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.5 Memory Management Functions

4.14.5.1 SGetSMem

Name

Get (allocate) a static memory pool.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetSMem (region, size, pool)
Region   region;
Size     size;
Pool     *pool;
```

Parameters

region

Region ID assigned by the system services at **xxActvInit** time.
Allowable values: 0 - 255.

size

Requested size, in bytes, of pool to allocate.

pool

Pointer to pool ID of allocated memory pool.

Description

This function allocates a static memory pool of the specified size within the specified memory region. After allocation, **SGetSBuf** and **SPutSBuf** may be used within the static pool to allocate buffers for structures needed by the task.

Typically, this function is called once by a task after it is configured by the layer manager with the maximums it needs to support.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.5.2 SPutSMem

Name

Put (deallocate) a static memory pool.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutSMem(region, pool)
Region   region;
Pool     pool;
```

Parameters

region

Region ID of memory region to which the memory pool must be returned. Allowable values: 0 - 255.

pool

Pool ID of memory pool being deallocated.

Description

This function deallocates a static memory pool within the specified memory region.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.5.3 SGetSBuf

Name

Get (allocate) a static memory buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetSBuf(region, pool, bufPtr, size)
Region   region;
Pool     pool;
Data     **bufPtr;
Size     size;
```

Parameters

region

Region ID of memory region from which to allocate the buffer.
Allowable values: 0 - 255.

pool

Pool ID of memory pool from which to allocate the buffer.
Allowable values: 0 - 255.

bufPtr

Pointer to the location where the allocated buffer pointer is placed.

size

Requested size, in bytes, of buffer to be allocated.

Description

This function allocates a static buffer of the specified size from the specified static memory pool.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.5.4 SPutSBuf

Name

Put (deallocate) a static memory buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutSBuf(region, pool, buf, size)
Region region;
Pool   pool;
Data   *buf;
Size   size;
```

Parameters

region

Region ID of memory region into which deallocated buffer must be returned. Allowable values: 0 - 255.

pool

Pool ID of memory pool into which deallocated buffer must be returned. Allowable values: 0 - 255.

buf

Pointer to the buffer to be returned.

size

Size, in bytes, of buffer to be returned. Size must agree with size requested in **SGetSBuf**.

Description

This function deallocates a buffer of the specified size back to the specified static memory pool.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.6 Message Functions

4.14.6.1 SGetMsg

Name

Get (allocate) a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetMsg(region, pool, mBufPtr)
Region   region;
Pool     pool;
Buffer   **mBufPtr;
```

Parameters

`region`

Region ID of memory region from which to allocate the message.
Allowable values: 0 - 255.

`pool`

Pool ID of memory pool from which to allocate the message.
Allowable values: 0 - 55.

`mBufPtr`

Pointer to the location where the allocated message pointer is placed.

Description

This function allocates a message from the specified dynamic memory pool and initializes its data contents to be empty.

Returns

ROK OK.

RFAILED Failed: Out of resources, illegal parameters, and so on.

4.14.6.2 SPutMsg

Name

Put (deallocate) a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutMsg(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Pointer to message buffer.

Description

This function deallocates a message. All data attached to the message is returned to the dynamic memory pool from which it was allocated. The message pointer is no longer valid.

Returns

ROK OK.

RFAILED Failed: Illegal parameters, and so on.

4.14.6.3 SInitMsg

Name

Initialize a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SInitMsg(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Pointer to message buffer.

Description

This function re-initializes the message so that its data contents are set to empty. All data attached to the message is returned to the dynamic memory pool from which it was allocated. The message pointer remains valid and unchanged.

Returns

ROK OK.

RFAILED Failed: Illegal parameters, and so on.

4.14.6.4 SFndLenMsg

Name

Find length of (the data contents of) a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SFndLenMsg(mBuf, lngPtr)
Buffer   *mBuf;
MsgLen   *lngPtr;
```

Parameters

mBuf

Pointer to message buffer.

lngPtr

Pointer to the location where the length of the message is placed.

Description

This function determines the length of the data contents of a message, and places the length count in the specified location. Message is unchanged.

Returns

ROK OK.

RFAILED Failed: Illegal parameters, and so on.

4.14.6.5 SExamMsg

Name

Examine (read a data byte in) a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SExamMsg(dataPtr, mBuf, idx)
Data     *dataPtr;
Buffer   *mBuf;
MsgLen   idx;
```

Parameters

dataPtr

Pointer to the location where the data byte is placed.

mBuf

Pointer to message buffer.

idx

Index into message (zero-based) where data reads from.

Description

This function reads one byte of data from a message at the specified index and places it in the specified location. The index is zero-based, that is, index value 0 indicates the first data byte of the message. The message remains unchanged.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (message too short).

RFAILED Failed: Illegal parameters, and so on.

4.14.6.6 SRepMsg

Name

Replace (a data byte in) a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRepMsg(data, mBuf, idx)
Data   data;
Buffer *mBuf;
MsgLen idx;
```

Parameters

data

Data byte that replaces existing data in the message.

mBuf

Pointer to message buffer.

idx

Index into message (zero-based) where data is replaced.

Description

This function replaces one byte of data at the specified index in a message. Message length is unchanged. The index is zero-based, that is, index value 0 indicates the first data byte of the message. Messages that have fewer data bytes than indicated in the index, remain unchanged.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (message too short).

RFAILED Failed: Illegal parameters, and so on.

4.14.6.7 SAddPreMsg

Name

Add one byte of data to the beginning of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddPreMsg(data, mBuf)
Data     data;
Buffer   *mBuf;
```

Parameters

data

One byte of data to be added to the beginning of a message.

mBuf

Pointer to message buffer.

Description

This function copies one byte of data and adds it to the beginning of a message. Message length is incremental by one. Additional memory buffers may need to be allocated and attached to the message to accommodate the new data.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.8 SAddPstMsg

Name

Add one byte of data to the end of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddPstMsg (data, mBuf)
Data     data;
Buffer   *mBuf;
```

Parameters

data

One byte of data to be added to the end of the message.

mBuf

Pointer to the message.

Description

This function copies one byte of data and adds it to the end of a message. Message length is incremental by one. Additional memory buffers may need to be allocated and attached to the message to accommodate the new data.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.9 RemPreMsg

Name

Remove one byte of date from the beginning of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRemPreMsg(dataPtr, mBuf)
Data     *dataPtr;
Buffer   *mBuf;
```

Parameters

dataPtr

Pointer to the location where one byte of data is placed.

mBuf

Message Buffer.

Description

This function removes one byte of data from the beginning of a message and puts it into the specified location. Message length decrements by one. Empty messages remain unchanged.

Returns

ROK OK: Removal successful, data is in specified location.

ROKDNA Failed: No data is available (message empty).

RFAILED Failed: Error.

4.14.6.10 RemPstMsg

Name

Remove one byte of data from the end of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRemPstMsg(dataPtr, mBuf)
Data     *dataPtr;
Buffer   *mBuf;
```

Parameters

dataPtr

Pointer to the location where the one byte of data is placed.

mBuf

Message Buffer.

Description

This function removes one byte of data from the end of a message and puts it into the specified location. Message length decrements by one. Empty messages remain unchanged.

Returns

ROK OK: Removal successful, data is in specified location.

ROKDNA Failed: No data is available (message empty).

RFAILED Failed: Error.

4.14.6.11 SAddPreMsgMult

Name

Add multiple bytes of data to the beginning of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddPreMsgMult(src, cnt, mBuf)
Data     *src;
MsgLen   cnt;
Buffer   *mBuf;
```

Parameters

src

Pointer to a string of data bytes to be added to the beginning of the message.

cnt

Number of bytes of data to add to the message.

mBuf

Message Buffer.

Description

This function copies the specified number of bytes of data and adds them to the beginning of a message. Message length is incremental by the specified count.

Additional memory buffers may need to be allocated and attached to the message to accommodate the new data.

Note: Upon addition to the message, the order of the data bytes is reversed, that is, the last byte of the data string becomes the first byte of the message. For example, if a data string "1234" is added to the message "says," the new message contents are "4321xyz."

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.12 SAddPstMsgMult

Name

Add multiple bytes of data to the end of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddPstMsgMult(src, cnt, mBuf)
Data     *src;
MsgLen   cnt;
Buffer   *mBuf;
```

Parameters

src

Pointer to a string of data bytes to be added to the end of the message.

cnt

Number of bytes of data to add to the message.

mBuf

Message Buffer.

Description

This function copies the specified number of bytes of data and adds them to the end of a message. Message length is incremental by the specified count. Additional memory buffers may need to be allocated and attached to the message to accommodate the new data.

Note: Upon addition to the message, the order of the data bytes is preserved to the message, that is, the last byte of the data string becomes the last byte of the message. For example, if a data string "1234" is added to the message "says," the new message contents are "xyz1234".

Returns

ROK OK.

RFAILED Failed: Error.

4.14.6.13 SGetPstMsgMult

Name

Allocate multiple bytes of data at the end of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetPstMsgMult(cnt, mBuf)
MsgLen cnt;
Buffer *mBuf;
```

Parameters

cnt

Number of bytes to be allocated at the end of the message.

mBuf

Pointer to the message.

Description

This function allocates storage for the specified number of data bytes at the end of a message and sets the contents of the new storage to 0. Message length is incriminated by the specified count.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.14 SRemPreMsgMult

Name

Remove multiple bytes of data from the beginning of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRemPreMsgMult(dst, cnt, mBuf)
Data     *dst;
MsgLen   cnt;
Buffer   *mBuf;
```

Parameters

dst

Pointer to the location where the data bytes are placed.

cnt

Number of bytes to be removed from the message.

mBuf

Pointer to the message.

Description

This function removes the specified number of bytes of data from the beginning of a message and puts them into the specified location. Message length decrements by the specified count. Messages that have less than the specified number of data bytes remain unchanged.

Note: Upon removal of the message, the order of the data bytes is preserved from the message--that is, the first byte of the message becomes the first byte of the data string. For example, if 4 bytes are removed from the message "1234xyz," the new message is "says" and the data string obtained is "1234."

Returns

ROK OK: Removal successful, data bytes are in specified location.

ROKDNA Failed: Specified number of data bytes is unavailable (message too short).

RFAILED Failed: Error.

4.14.6.15 SRemPstMsgMult

Name

Remove multiple bytes of data from the end of a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRemPstMsgMult(dst, cnt, mBuf)
Data     *dst;
MsgLen   cnt;
Buffer   *mBuf;
```

Parameters

dst

Pointer to the location where the data bytes are placed.

cnt

Number of bytes to be removed from the message.

mBuf

Message Buffer.

Description

This function removes the specified number of bytes of data from the end of a message and puts them into the specified location. Message length decrements by the specified count. Messages that have less than the specified number of data bytes remain unchanged.

Note: Upon removal of the message, the order of the data bytes is reversed from the message--that is, the last byte of the message becomes the first byte of the data string. For example, if 4 bytes are removed from the message "xyz1234," the new message is "says" and the data string obtained is "4321."

Returns

ROK OK: Removal successful, data bytes are in specified location.

ROKDNA Failed: Specified number of data bytes is unavailable (message too short).

RFAILED Failed: Error.

4.14.6.16 SCpyFixMsg

Name

Copy from multiple bytes of contiguous data to a message at a specified index.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCpyFixMsg(srcBuf, dstMbuf, dstIdx, cnt, cCnt)
Data     *srcBuf;
Buffer   *dstMbuf;
MsgLen   dstIdx;
MsgLen   cnt;
MsgLen   *cCnt;
```

Parameters

`srcBuf`

Pointer to a string of data bytes (fixed buffer) to be added to the message.

`dstMbuf`

Pointer to the message.

`dstIdx`

Index in the message (zero-based) where data bytes are to be added.

`cnt`

Number of bytes of data to add to the message.

`cCnt`

Pointer to location where the count of number of bytes of data actually added to the message is placed.

This function copies the specified number of bytes of data (from the fixed buffer) and adds them to the message at the specified index. The index is zero-based, that is, index value 0 indicates the first data byte of the message. Message length is incremental by the specified count. Additional memory buffers may need to be allocated and attached to the message to accommodate the new data. The count of number of data bytes added is returned in the specified location.

Note: If the index is 0, upon addition to the message, the order of the data bytes is reversed-- that is, the last byte of the data string becomes the first byte of the message. For example, if a data string "1234" is added to the message "says" at index 0, the new message contents are "4321xyz."

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.17 SCpyMsgFix

Name

Copy multiple bytes of data from a message at a specified index into a contiguous buffer.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCpyMsgFix(srcMbuf, srcIdx, cnt, dstBuf, cCnt)
Buffer   *srcMbuf;
MsgLen   srcIdx;
MsgLen   cnt;
Data     *dstBuf;
MsgLen   *cCnt;
```

Parameters

srcMbuf

Pointer to the message.

srcIdx

Index in the message (zero-based) from where data bytes are to be copied.

cnt

Number of bytes of data to be copied from the message.

dstBuf

Pointer to the location (fixed buffer) where the copied data bytes is placed.

cCnt

Pointer to location where the count of number of bytes of data actually copied from the message is placed.

Description

This function copies the specified number of bytes of data from the message at the specified index and puts them into the specified location. The index is zero-based, that is, index value 0 indicates the first data byte of the message. Message is unchanged.

Note: Upon removal from the message, the order of the data bytes is preserved from the message. For example, if 4 bytes are copied from the message "1234xyz" at index 2, the data string obtained is "34xy."

Returns

ROK OK: Copy successful, data bytes are in specified location.

ROKDNA Failed: Specified index or specified number of bytes not available (message too short).

RFAILED Failed: Error.

4.14.6.18 SCpyMsgMsg

Name

Copy from message to new message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCpyMsgMsg(srcBuf, region, pool, dstBuf)
Buffer   *srcBuf;
Region   dstRegion;
Pool     dstPool;
Buffer   **dstBuf;
```

Parameters

srcBuf

Pointer to source message.

dstRegion

Region ID of memory region from which to allocate the new message.
Allowable values: 0 - 255.

dstPool

Pool ID of memory pool from which to allocate the new message.
Allowable values: 0 - 255.

dstBuf

Pointer to the location where the newly allocated message pointer is placed.

Description

This function allocates a new message in the specified memory region and pool, copies the data contents of the source message into the new message, and returns the new message pointer at the specified location.

Returns

ROK OK.

ROUTRES Failed: Out of resources for allocating new message.

RFAILED Failed: Error.

4.14.6.19 SCatMsg

Name

Concatenate two messages.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCatMsg(mBuf1, mBuf2, order)
Buffer   *mBuf1;
Buffer   *mBuf2;
Order    order;
```

Parameters

`mBuf1`

Pointer to message buffer 1.

`mBuf2`

Pointer to message buffer 2.

`order`

Order in which messages are concatenated. Allowable values are:

- M1M2: Place message 2 at the end of message 1.
- M2M1: Place message 2 in front of message 1.

Description

This function concatenates the two specified messages into one message. If the specified order is **`M1M2`**, all data attached to message 2 is moved to the end of message 1. If the specified order is **`M2M1`**, all data attached to message 2 is moved to the front of message 1.

Message 2 is set to empty. The length of message 1 length is increased by the length of message 2. The length of message 2 is set to zero. Message 2 is not deallocated.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.20 SSegMsg

Name

Segment a message into two messages.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSegMsg(mBuf1, idx, mBuf2)
Buffer *mBuf1;
MsgLen idx;
Buffer **mBuf2;
```

Parameters

mBuf1

Pointer to message buffer 1 (original message to be segmented).

idx

Index into message 1 (zero-based) from which message 2 is created.

mBuf2

Pointer to message buffer 2 (new message).

Description

This function segments message 1 into two messages at the specified index. The index is zero-based, that is, index value 0 indicates the first data byte of message 1.

If the index is less than the length of message 1, message 2 is created. All data attached to message 1 from the index (inclusive) is moved to message 2. The length of message 1 is set to the index value.

If the index is greater than the length of message 1, message 1 remains unchanged. Message 2 is set to null.

If message buffer two is not set to null, system services is responsible for allocating the second message buffer.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (message 1 too short).

ROUTERES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.21 SCompressMsg

Name

Compress message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCompressMsg(mBuf)
Buffer *mBuf;
```

Parameters

mBuf

Pointer to message buffer.

Description

This function compresses the storage requirements for the data contents of a message such that the fewest possible number of dynamic buffers are used. This does not affect the data contents of the message. No compression algorithm is applied to the data contents; only the storage requirements are affected.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.6.22 SAddMsgRef

Name

If possible, increment the reference count of the number of users on an existing message.

Direction

Layer software to system services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddMsgRef(srcBuf, dstRegion, dstPool, dstBuf)
Buffer *srcBuf;
Region dstRegion;
Pool   dstPool;
Buffer **dstBuf;
```

Parameters

`srcBuf`

Pointer to the existing message.

`dstRegion`

Region ID of memory region from which to allocate the reference message. Allowable values: 0 - 255.

`dstPool`

Pool ID of memory pool from which to allocate the reference message. Allowable values: 0 - 255.

`dstBuf`

Pointer to the location where the allocated message reference pointer is placed.

Description

This function allocates a message from the specified dynamic region and pool, and initializes this new message to refer to the data contents of the original message.

The reference count to the original message increments to indicate that the data portion of the message is now being shared.

Any change made to the data contents of the original message, after a call to this function, must not be reflected across to the new message and vice versa. To keep changes to the data portion transparent, it may be necessary to duplicate the data portion of the message. In such cases, the reference count in the original message decrements to reflect the new status of the message.

Note: If the **`dstRegion`** and **`dstPool`** are not identical to the region and pool of the existing message, then the data portion of the existing message must be duplicated in the new message. The reference count to the original message does not increment.

Returns

ROK OK.

RFAILED Failed: Error.

ROUTRES Failed: Out of resources.

4.14.6.23 SPkS8

Name

Pack (add) a signed 8-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkS8(val, mBuf)
S8       val;
Buffer   *mBuf;
```

Parameters

val

Data to be added to the message.

mBuf

Pointer to message buffer.

Description

This function adds one byte of data (signed 8-bit value) to the beginning of a message.

The code for the body of this function is provided and makes calls to other system service primitives.

The function **SUnpkS8** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.24 SPkU8

Name

Pack (add) an unsigned 8-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkU8(val, mBuf)
U8       val;
Buffer   *mBuf;
```

Parameters

val

Data to be added to the message.

mBuf

Pointer to message buffer.

Description

This function adds one byte of data (unsigned 8-bit value) to the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **SUnpkU8** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.25 SPkS16

Name

Pack (add) a signed 16-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkS16(val, mBuf)
S16      val;
Buffer   *mBuf;
```

Parameters

val

Data to be added to the message.

mBuf

Pointer to message buffer.

Description

This function adds two bytes of data (signed 16-bit value) to the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **SUnpkS16** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.26 SPkU16

Name

Pack (add) an unsigned 16-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkU16(val, mBuf)
U16      val;
Buffer   *mBuf;
```

Parameters

`val`

Data to be added to the message.

`mBuf`

Pointer to message buffer.

Description

This function adds two bytes of data (unsigned 16-bit value) to the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SUnpkU16`** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.27 SPkS32

Name

Pack (add) a signed 32-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkS32(val, mBuf)
S32      val;
Buffer   *mBuf;
```

Parameters

`val`

Data to be added to the message.

`mBuf`

Pointer to message buffer.

Description

This function adds four bytes of data (signed 32-bit value) to the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SUnpkS32`** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.28 SPkU32

Name

Pack (add) an unsigned 16-bit value to a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPkU32(val, mBuf)
U32      val;
Buffer   *mBuf;
```

Parameters

`val`

Data to be added to the message.

`mBuf`

Pointer to message buffer.

Description

This function adds four bytes of data (unsigned 32-bit value) to the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SUnpkU32`** must be used to unpack the data from a message packed by this function.

Returns

ROK OK.

ROUTRES Failed: Out of resources.

RFAILED Failed: Error.

4.14.6.29 SUnpkS8

Name

Unpack (remove) a signed 8-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkS8(val, mBuf)
S8     *val;
Buffer *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes one byte of data (signed 8-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkS8`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.6.30 SUnpkU8

Name

Unpack (remove) an unsigned 8-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkU8(val, mBuf)
U8       *val;
Buffer   *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes one byte of data (unsigned 8-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkU8`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.6.31 SUnpkS16

Name

Unpack (remove) a signed 16-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkS16(val, mBuf)
S16      *val;
Buffer   *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes two bytes of data (signed 16-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkS16`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.6.32 SUnpkU16

Name

Unpack (remove) an unsigned 16-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkU16(val, mBuf)
U16      *val;
Buffer   *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes two bytes of data (unsigned 16-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkU16`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.6.33 SUnpkS32

Name

Unpack (remove) a signed 32-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkS32(val, mBuf)
S32      *val;
Buffer   *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes four bytes of data (signed 32-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkS32`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.6.34 SUnpkU32

Name

Unpack (remove) an unsigned 32-bit value from a message.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: Yes.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnpkU32(val, mBuf)
U32      *val;
Buffer   *mBuf;
```

Parameters

`val`

Pointer to location where data removed from the message is placed.

`mBuf`

Pointer to message buffer.

Description

This function removes four bytes of data (unsigned 32-bit value) from the beginning of a message. The code for the body of this function is provided and makes calls to other system service primitives.

The function **`SPkU32`** must be used to pack the data to a message unpacked by this function.

Returns

ROK OK.

ROKDNA OK: Data not available

RFAILED Failed: Error.

4.14.7 Queue Management Functions

4.14.7.1 SInitQueue

Name

Initialize a queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SInitQueue(q)
Queue *q;
```

Parameters

q

Pointer to the queue.

Description

This function initializes the queue. After initialization, any other queue function may be used to manipulate the queue. Queue length is set to zero.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.2 SQueueFirst

Name

Queue a message at the beginning of queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SQueueFirst(buf, q)
Buffer   *buf;
Queue    *q;
```

Parameters

buf

Pointer to the message to be queued.

q

Pointer to the queue.

Description

This function queues a message at the beginning of the specified queue. Queue length is incremental by one.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.3 SQueueLast

Name

Queue a message at the end of a queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SQueueLast(buf, q)
Buffer   *buf;
Queue    *q;
```

Parameters

buf

Pointer to the message to be queued.

q

Pointer to the queue.

Description

This function queues a message at the end of the specified queue. Queue length is incremental by one.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.4 SDequeueFirst

Name

Dequeue a message from the beginning of queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SDequeueFirst(bufPtr, q)
Buffer   **bufPtr;
Queue    *q;
```

Parameters

bufPtr

Pointer to the location where the de-queued message is placed.

q

Pointer to the queue.

Description

This function de-queues a message from the beginning of the specified queue. Queue length decrements by one. Empty queues are unchanged.

Returns

ROK OK.

ROKDNA Failed: No message available (queue empty).

RFAILED Failed: Error.

4.14.7.5 SDequeueLast

Name

Dequeue a message from the end of queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SDequeueLast(bufPtr, q)
Buffer   **bufPtr;
Queue    *q;
```

Parameters

bufPtr

Pointer to the location where the de-queued message is placed.

q

Pointer to the queue.

Description

This function de-queues a message from the end of the specified queue. Queue length decrements by one. Empty queues are unchanged.

Returns

ROK OK.

ROKDNA Failed: No message available (queue empty).

RFAILED Failed: Error.

4.14.7.6 SFlushQueue

Name

Flushes the entire contents of a queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE S16 SFlushQueue(q)
Queue    *q;
```

Parameters

q

Pointer to the queue.

Description

This function de-queues all of the buffers from the specified queue. Queue length is set to zero. No action is taken if the queue is empty. If the de-queued buffer is a message buffer, all data buffers associated with the message buffer are returned to memory.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.7 SCatQueue

Name

Concatenate two queues.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCatQueue(q1, q2, order)
Queue *q1;
Queue *q2;
Order order;
```

Parameters

`q1`

Pointer to queue 1.

`q2`

Pointer to queue 2.

`order`

Order in which queues are concatenated. Allowable values are:

- **`Q1Q2`**: Place queue 2 at the end of queue 1.
- **`Q2Q1`**: Place queue 2 at the beginning of queue 1.

Description

This function concatenates the two specified queues into one queue. If the order is **`Q1Q2`**, all messages attached to queue 2 are moved to the end of queue 1. If the order is **`Q2Q1`**, all buffers attached to queue 2 are moved to the front of queue 1.

Queue 2 is set to empty. The length of queue 1 is increased by the length of queue 2.

Note: If the order is **`Q2Q1`**, upon addition to queue 1, the order of messages in queue 2 is reversed, that is, the last message in queue 2 becomes the first message in queue 1. For example, if queue 1 had messages "a1, a2, a3" and queue 2 had messages "b1, b2, b3," then the concatenated queue 1 have "b3, b2, b1, a1, a2, a3."

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.8 SFndLenQueue

Name

Find length of (number of messages in) a queue.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SFndLenQueue(q, lngPtr)
Queue    *q;
Qlen     *lngPtr;
```

Parameters

q

Pointer to the queue.

lngPtr

Pointer to the location where the length of the queue is placed.

Description

This function determines the length of a queue (that is, the number of messages in the queue) and returns it in the specified location.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.7.9 SAddQueue

Name

Add a message to a queue at a specified index.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SAddQueue(buf, q, idx)
Buffer   *buf;
Queue    *q;
QLen     idx;
```

Parameters

buf

Pointer to the message to be placed in the queue.

q

Pointer to the queue.

idx

Index into queue (zero-based) where message is placed in the queue.

Description

This function inserts the message into the queue at the specified index. The index is zero- based (that is, the index value 0 indicates the first position in the queue). Queue length is incremental by one.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (queue too short).

RFAILED Failed: Error.

4.14.7.10 SRemQueue

Name

Remove a message from a queue at a specified index.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRemQueue(bufPtr, q, idx)
Buffer   **bufPtr;
Queue    *q;
QLen     idx;
```

Parameters

bufPtr

Pointer to the location where the de-queued message is placed.

q

Pointer to the queue.

idx

Index into queue (zero-based) where queue is examined.

Description

This function removes a message from the queue at the specified index. The index is zero-based (that is, the index value 0 indicates the first position in the queue). Queue length decrements by one.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (queue too short).

RFAILED Failed: Error.

4.14.7.11 SExamQueue

Name

Examine (read a message from) a queue at a specified index.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SExamQueue(bufPtr, q, idx)
Buffer   **bufPtr;
Queue    *q;
QLen     idx;
```

Parameters

bufPtr

Pointer to the location where the de-queued message is placed.

q

Pointer to the queue.

idx

Index into queue (zero-based) where queue is examined.

Description

This function retrieves a message from the queue at the specified index. Index is zero-based (that is, the index value 0 indicates the first position in the queue). Queue remains unchanged.

Returns

ROK OK.

ROKDNA Failed: Specified index not available (queue too short).

RFAILED Failed: Error.

4.14.8 Miscellaneous Functions

4.14.8.1 SFndProcId

Name

Find processor ID.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC ProcId SFndProcId()
```

Parameters

None.

Description

This function finds the processor ID of the processor on which the calling task is running.

Returns

Local processor ID.

4.14.8.2 SSetProcId

Name

Set processor ID.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SSetProcId(pId)
ProcId pId;
```

Parameters

pId

Processor ID.

Description

This function sets the processor ID for the local processor.

Returns

None.

4.14.8.3 SSetDateTime

Name

Set Date and Time.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSetDateTime(dt)
DateTime *dt;
```

Parameters

dt

Pointer to date and time structure. Date and time structure has the following format:

```
typedef struct dateTime
{
   U8 month;
   U8 day;
   U8 year;
   U8 hour;
   U8 min;
   U8 sec;
   U8 tenths;
} DateTime;
```

month

Month. Allowable values: 1 - 12.

day

Day. Allowable values: 1 - 31.

year

Year. Allowable values: 0 - 99.

hour

Hour. Allowable values: 0 - 23.

min

Minute. Allowable values: 0 - 59.

sec

Second. Allowable values: 0 - 59.

tenths

Tenths of second. Allowable values: 0 - 9.

Description

This function is used to set the calendar date and time.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.4 SGetDateTime

Name

Get Date and Time.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetDateTime(dt)
DateTime *dt;
```

Parameters

dt

Date and time structure. If the return is OK, the contents of the **dateTime** structure are valid. If the return is not OK, the contents of the **dateTime** structure is zero. Date and time structure has the following format:

```
typedef struct dateTime
{
   U8 month;
   U8 day;
   U8 year;
   U8 hour;
   U8 min;
   U8 sec;
   U8 tenths;
} DateTime;
```

month

Month. Allowable values: 1 - 12.

day

Day. Allowable values: 1 - 31.

year

Year. Allowable values: 0 - 99.

hour

Hour. Allowable values: 0 - 23.

min

Minute. Allowable values: 0 - 59.

sec

Second. Allowable values: 0 - 59.

tenths

Tenths of second. Allowable values: 0 - 9.

Description

This function is used to determine the calendar date and time. This information may be used for some management functions.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.5 SGetSysTime

Name

Get System Time.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetSysTime(sysTime)
Ticks *sysTime;
```

Parameters

sysTime

Pointer to the location where the system time is placed.

Description

This function is used to determine the system time. This information may be used for some management functions. The routine returns the number of "ticks" accrued since the system started running (boot time). "Tick" resolution is system-dependent.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.6 SRandom

Name

Generate a pseudo random number.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRandom(value)
Random *value;
```

Parameters

value

Location where the generated pseudo random number is placed.

Description

This function generates a pseudo random number. Refer to the article "Generating and Testing Pseudorandom Numbers" by Charles Whiten in the October 1984 issue of Byte magazine for a description of the implementation of the pseudo random number generator.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.7 SError

Name

Handle unrecoverable software error.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SError(seq, reason)
Seq    seq;
Reason reason;
```

Parameters

seq

Sequence number indicates specific location in software from which the function was called.

reason

Reason indicates specific software error detected.

Description

Invoked by layer when a unrecoverable software error is detected. This function must never return.

Note: This routine is in the process of being phased out. Refer to the description of **SLogError** for a description of the preferred interface for handling errors

---

Returns

RFAILED Failed.

4.14.8.8 SLogError

Name

Log a software error.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SLogError(ent, inst, procId, file, line, errCls,
                      errCode, errVal, errDesc)
Ent     ent;
Inst    inst;
ProcId  procId;
Txt     *file;
S32     line;
ErrCls  errCls;
ErrCode errCode;
ErrVal  errVal;
Txt     *errDesc;
```

Parameters

ent

Entity ID of the calling task.

inst

Instance of the entity.

procId

Processor ID.

file

Source code file name from where the call is made (**usually __FILE__**).

line

Source code line number from where the call is made (**usually __LINE__**).

errCls

Error Class - defined in **ssi.h**.

Allowable values:

0x1 ERRCLS_ADD_RES - error class "add resources" (refer to section on error checking).

0x2 ERRCLS_INT_PAR - error class "interface parameter" (refer to section on error checking).

0x1 ERRCLS_DEBUG - error class "debug" (refer to section on error checking).

errCode

Layer unique error code.

errVal

Error Value.

errDesc

Textual description of error.

Description

Invoked by layer when a software error is detected. This function is expected to halt the system if the **errCls** passed to it is **ERRCLS_DEBUG**. Refer to the section on Error Checking and Recovery for more information of the use of this primitive.

Returns

RFAILED Failed.

4.14.8.9 SChkRes

Name

Check (memory buffer) resources available in a memory pool.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SChkRes(region, pool, status)
Region region;
Pool   pool;
S16    *status;
```

Parameters

region

Region ID of memory region whose resources are to be checked.

pool

Pool ID of memory pool whose resources are to be checked.

status

Pointer to address where the resource status is placed. Allowable values: 0 - 10. A value of 10 indicates 100% of memory is available, a value of 5 indicates 50% of memory is available, and so on.

Description

This function is used to check available buffers in the specified memory region and pool.

Note: Application can utilize this API to check the memory availability status as part of the Admission Control, and can take appropriate decision for admitting the call/dialogue/transaction. Hence, some of the Trillium layers might not use this API.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.10 SPrint

Name

Print a pre-formatted character string to default output device.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC SPrint(buf)
Txt *buf;
```

Parameters

buf

Pre-formatted character string to be printed. It is terminated by a null '\0' character.

Description

This function prints a pre-formatted, null-terminated character string. Typical usage consists of a call to **sprintf** to format the string, followed by a call to **SPrint.**

SPrint is replaced by **SDisplay**.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.11 SDisplay

Name

Display a pre-formatted character string on a specific output device.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC SDisplay(chan, buf)
S16 chan;
Txt *buf;
```

Parameters

chan

Output device handle. Channel 0 is reserved for backwards compatibility with **SPrint**.

buf

Pre-formatted character string to be printed. It is terminated by a null '\0' character.

Description

This function prints a pre-formatted, null-terminated character string to a specified output device. Typical usage consists of a call to **`sprintf`** to format the string, followed by a call to **`SDisplay`**.

`SDisplay` replaces **`SPrint.`**

Returns

ROK OK.

RFAILED Failed: Error.

4.14.8.12 SPrntMsg

Name

Print the contents of a message to default output device.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPrntMsg(mBuf, src, dst)
Buffer *mBuf;
S16 src;
S16 dst;
```

Parameters

`mBuf`

Message to be printed.

`src`

Source ID.

`dst`

Destination ID.

Description

This function prints the following information for a message: queue length, message length, direction, hexadecimal, and ASCII (if appropriate) values of all data bytes in the message.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9 Multi-threaded System Service Primitives

These routines are available only for system service implementations that support threads (-DMT).

4.14.9.1 SGetMutex

Name

Allocate a mutex.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetMutex(mId)
MtMtxId *mId;
```

Parameters

mId

Pointer to mutex ID (returned).

Description

This functions allocates a mutex.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.2 SPutMutex

Name

Deallocate a mutex.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutMutex(mId)
MtMtxId mId;
```

Parameters

mId

Mutex ID to deallocate.

Description

This function deallocates a mutex.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.3 SLockMutex

Name

Lock a mutex.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SLockMutex(mId)
MtMtxId mId;
```

Parameters

mId

Mutex ID.

Description

This function locks a mutex.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.4 SUnlockMutex

Name

Unlock a mutex.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SUnlockMutex(mId)
MtMtxId mId;
```

Parameters

mId

Mutex ID.

Description

This function unlocks a mutex.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.5 SGetCond

Name

Allocate a conditional variable.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetCond(cId)
MtCondId *cId;
```

Parameters

cId

Pointer to condition variable ID (returned).

Description

This function allocates a condition variable.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.6 SPutCond

Name

Deallocate a condition variable.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutCond(cId)
MtCondId *cId;
```

Parameters

cId

Condition variable ID.

Description

This function deallocates a condition variable.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.7 SCondWait

Name

Wait of condition variable.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCondWait(mId, cId)
MtMtxId  mId;
MtCondId cId;
```

Parameters

mId

Mutex ID.

cId

Condition variable ID.

Description

This function unlocks the specified **mutex** and sleeps until a signal is sent on the specified condition variable (refer to **SCondSignal** and **SCondBroadcast**). Upon receipt of a signal, **SCondWait** reacquires the lock on the specified **mutex** and returns.

Example: i
```
f (SLockMutex(tsk->dq.mtx) != ROK)
continue;
while(tsk->dq.bufQ.crntSize == 0) /* demand queue is empty */
SCondWait(tsk->dq.mtx, tsk->dq.cond);

/*
 * condition variable signal received, continue.
 * do work...
 */
.
.
.
  SUnlockMutex(tsk->dq.mtx); /* release the demand queue */
```

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.8 SCondSignal

Name

Signal a condition variable.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCondSignal(cId)
MtCondId cId;
```

Parameters

cId

Condition variable ID.

Description

This function generates a signal to a single thread waiting on a condition variable.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.9 SCondBroadcast

Name

Broadcast a signal to all threads waiting on a condition variable.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SCondBroadcast(cId)
MtCondId *cId;
```

Parameters

cId

Condition variable ID.

Description

This function broadcasts a signal to all threads waiting on a condition variable.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.10 SGetThread

Name

Allocate a new thread of control.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetThread(thrd, thr_flags, arg, thrdId)
MtThrd      thrd;
MtThrdFlags thr_flags;
Ptr         arg,
MtThrdId    *thrdId;
```

Parameters

thrd

Thread entry point function. This function acts as the "main" function for this new thread.

thr_flags

Flags used to create the threads. Allowable values are:

0x00 MT_THR_NOFLAGS - no flags.

0x01 **MT_THR_SUSPENDED** - initialize thread in a suspended state.

0x02 **MT_THR_DETACHED** - initialize thread detached (no lwp affinity).

0x04 **MT_THR_BOUND** - initialize thread bound (to a lwp).

0x08 **MT_THR_NEW_LWP** - add new lwp to lwp pool (bump concurrency).

0x10 **MT_THR_DAEMON** - initialize thread to run as a daemon (allow main thread to exit).

arg

Argument to be passed into the thread at invocation (cast as **void ***).

thrdId

Thread ID (returned).

Description

This function instantiates a new thread of control.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.11 SPutThread

Name

Deallocate thread.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutThread(thrdId)
MtThrdId thrdId;
```

Parameters

thrdId

Thread ID.

Description

This function deallocates a thread.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.9.12 SThreadYield

Name

Yield to another thread.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC INLINE Void SThreadYield()
```

Parameters

None.

Description

This function causes the calling thread to yield control to another thread.

Returns

None.

4.14.9.13 SThreadExit

Name

Cause a thread to exit.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SThreadExit(status)
Ptr *status;
```

Parameters

`status`

Pointer to exit status.

Description

This function causes the calling thread to exit.

Returns

None.

4.14.9.14 SSetThrdPrior

Name

Set thread priority.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SSetThrdPrior(tId, tPr)
MtThrdId tId;
MtThrdPrior tPr;
```

Parameters

tId

Thread ID.

tPr

Thread priority. Allowable values are:

- **0 MT_LOW_PRIOR** - lowest priority.
- **10 MT_NORM_PRIOR** - normal priority.
- **20 MT_HIGH_PRIOR** - highest priority.

Description

This function sets the scheduling priority of the specified thread.

Returns

None.

4.14.9.15 SGetThrdPrior

Name

Get thread priority.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SGetThrdPrior(tId, tPr)
MtThrdId    tId;
MtThrdPrior *tPr;
```

Parameters

tId

Thread ID.

tPr

Thread priority (returned). Allowable values are:

- **0 MT_LOW_PRIOR** - lowest priority.
- **10 MT_NORM_PRIOR** - normal priority.
- **20 MT_HIGH_PRIOR** - highest priority.

Description

This function returns the scheduling priority of the specified thread.

Returns

None.

4.14.9.16 SExit

Name

Gracefully exit the process.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC Void SExit()
```

Parameters

None.

Description

This function gracefully exits the process.

Returns

None.

4.14.10 Microsoft Windows NT Kernel Primitives

4.14.10.1 SPutIsrDpr

Name

Install interrupt service routine and deferred procedure call.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SPutIsrDpr(vectNmb, context, isrFnct, dprFnct)
VectNmb vectNmb;
Void    *context;
PIF     isrFnct;
PIF     dprFnct;
```

Parameters

vectNmb

Vector number.

context

Context.

isrFnct

Interrupt service routine.

dprFnct

Deferred procedure call.

Description

Install an interrupt service routine and deferred procedure call pair for the specified vector number.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.10.2 SSyncInt

Name

Synchronize interrupt.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSyncInt(adaptorNmb, syncFnct, syncContext)
U16    adaptorNmb;
PFVOID syncFnct;
Void   *syncContext;
```

Parameters

adaptorNmb

Adaptor Number.

syncFnct

Synchronization function.

syncContext

Synchronization context.

Description

Synchronize interrupt with the specific adaptor.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.11 Multi-core Support Functions

4.14.11.1 SRegCpuInfo

Name

Register the CPU core information with SSI.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SRegCpuInfo(cpuInfo)
SCpuInfo*     cpuInfo;
```

Parameters

`cpuInfo`

The structure containing the information about the number of cores and threads per core.

Description

Register the information related to the CPU cores available for SSI and the number of threads available for used by SSI.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.11.2 SSetAffinity

Name

Set processor affinity for a system task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SSetAffinity(tskId, mode, coreId, tskAssociatedTskId)
SSTskId*           tskId;
SsAffinityMode     mode;
U32                coreId;
SSTskId*           tskAssociatedTskId;
```

Parameters

`tskId`

The task ID of the system task for which affinity has to be set.

`mode`

The mode for setting the processor affinity.

`coreId`

The ID of the core to which the affinity has to be set.

`tskAssociatedTskId`

The task ID of another system task which runs on the core with ID as "**`coreId`**".

Description

Set processor/core affinity for the system task identified by **`tskId`** on to the core with ID as **`coreId`**. The function takes a "mode" argument which is used to set the affinity based on users preferences. The available modes are:

- **`SS_AFFINITY_MODE_DEFAULT`**
- **`SS_AFFINITY_MODE_SPECIFIC`**
- **`SS_AFFINITY_MODE_ASSOC`**
- **`SS_AFFINITY_MODE_EXCL`**

The first mode "**SS_AFFINITY_MODE_DEFAULT**" is used to leave the intelligence of setting affinity to an appropriate core, based on the information registered using **SRegCpuInfo()** function. SSI assigns system tasks to the cores in a round-robin fashion. If all the available threads on a core are assigned to different tasks then a warning message is given. The "SPECIFIC" mode is used when the user wants to override the SSI assignment and wants to assign a system task to the users desired core. The user can assign two tasks on the same core to get the advantage of cache sharing by using the "ASSOC" mode. In this mode the user has to give the ID of the system task which must be run on the same core as "**tskAssociatedTskId**". The last mode is "EXCL" which is used to set a particular system task as exclusive so that SSI does not assign any other system task the same core ID.

Returns

ROK OK.

RFAILED Failed: Error.

4.14.11.3 SGetAffinity

Name

Get affinity information for a system task.

Direction

Layer Software to System Services.

Supplied

- Protocol layer product: No.
- System services product: Yes.

Synopsis

```
PUBLIC S16 SGetAffinity(tskId, coreId)
SSTskId*   tskId;
U32*       coreId;
```

Parameters

tskId

The task ID of the system task for which affinity is retrieved.

coreId

The ID of the core to which the affinity is set.

Description

This function is used by the user to get the processor affinity information for the system task "**tskId**".

Returns

ROK OK.

RFAILED Failed: Error.

4.14.11.4 SGetThrdProf

Name

Get the current thread profiling information for a system task.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SGetThrdProf(sTskId, procId, ent, inst, curEvnt,
curEvtTime, totTime)
SSTskId     *sTskId;
ProcId      procId;
Ent         ent;
Inst        inst;
Event       *curEvnt;
U32         *curEvtTime;
U32         *totTime;
```

Parameters

sTskId

The task ID of the system task for which the thread profiling information must be retrieved.

procId

Processor ID.

ent

Entity ID.

inst

Instance ID.

curEvnt

The current event.

curEvtTime

Time taken to execute last scheduled operation.

totTime

Total time spent in the current entity.

Description

This function is used by the user to get the thread profile information for a specific TAPA task. It returns the current values of entity, instance, event, and time taken for the thread when it was last scheduled or during some operation to perform.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.5 SRegCfgWd

Name

Configure the watchdog functionality.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SRegCfgWd(numNodes, addr, port, timeout, callback, data)
U32             numNodes;
U8              *addr[];
U16             port[];
U32             timeout;
WdUserCallback  callback;
void            *data;
```

Parameters

numNodes

The number of nodes to be configured for watchdog functionality.

addr

The list of IP addresses of the nodes to be configured for watchdog functionality.

port

The list of ports for the nodes to be configured for watchdog functionality.

timeout

Heartbeat time-out.

callback

The user callback to be called for heartbeat failure.

timetaken

The user data to be passed back to the user callback.

Description

This function is used by the user to configure the watchdog functionality on the node.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.6 SDeregCfgWd

Name

Unconfigure the watchdog functionality.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SDeregCfgWd(void)
```

Parameters

None.

Description

This function is used by the user to unconfigure the watchdog functionality on the node.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.7 SStartHrtBt

Name

Initiate the heartbeat mechanism on the node.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SStartHrtBt(timeInterval)
U8              timeInterval;
```

Parameters

timeInterval

The heartbeat timer time-out.

Description

This function is used by the user to start the heartbeat mechanism with all the configured nodes.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.8 SStopHrtBt

Name

Stop the heartbeat mechanism on the node.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SStopHrtBt(void)
```

Parameters

None.

Description

This function is used by the user to stop the heartbeat mechanism with all the configured nodes.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.9 SRegLogCfg

Name

Configure the SSI Logger.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SRegLogCfg(mode, path, size, ipaddress, port)
U8              mode;
S8              *path;
U32             size;
S8              *ipaddress;
U16             port;
```

Parameters

mode

Mode of the log file to be opened.

path

Path of the log file.

size

Log file size limit.

ipaddress

IP address of the node for the logs to be streamed.

port

Port to which the logs must be streamed.

Description

This function is used by the user to configure the SSI logger.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.10 SDeregLogCfg

Name

Unconfigure the SSI Logger.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SDeregLogCfg(void)
```

Parameters

None.

Description

This function is used by the user to unconfigure and de-register the SSI logger task.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.11 SHstGrmInfoShow

Name

Get the memory histogram information.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SHstGrmInfoShow(ent)
Ent             *ent;
```

Parameters

ent

Entity ID.

Description

This function is used by the user to get the memory histogram information based on the TAPA task for the given entity ID.

Returns

ROK OK.

RFAILED Failed: error.

4.14.11.12 SLogLkInfo

Name

Log memory leak information.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC Void SLogLkInfo(void)
```

Parameters

None.

Description

This function is used to log the memory leak information.

Returns

Void.

4.14.11.13 SRegTskInfo

Name

Register the TAPA task information.

Direction

Layer Software to System Services.

Supplied

Protocol layer product: No.

System services product: Yes.

Synopsis

```
PUBLIC S16 SRegTskInfo(cfgFile)
U8* cfgFile;
```

Parameters

`cfgFile`

The configuration file containing the thread to core affinity information, and TAPA task to thread attach information.

Description

This function is used to register the TAPA tasks and set CPU affinity for the threads based on the configuration specified in the file **`cfgFile`**.

Returns

ROK OK.

RFAILED Failed: error.

5

Usage

5.1 Initialization

The initialization function is called by the system services to initialize a task. The layer management entity that is building the protocol stack on the system, registers the task initialization function with system services.

Task initialization must be done before a task can be scheduled for any event (timers, messages, and so on).

Sample of the code in the layer management entity to register the initialization function for a task with system services:

```
Ent  ent;
Inst inst;
ent = TSTENT;                          /* Entity ID of task */
inst = TSTINST0;                       /* Instance ID of task */
SRegInit(ent, inst, tstActvInit);      /* Register init function
                                          for task */
```

5.2 Timer Activation

The timer activation functions are called by system services periodically to activate a task, so that the task can manage its internal timers.

The timer activation functions are registered with system services by the task itself anytime after it is configured with general parameters.

Sample of the code in a task to register the timer activation functions:

```
Ent ent;
Inst inst;
ent = TSTENT;                              /* Entity ID of task */
inst = TSTINST0;                           /* Instance ID of task */
SRegTmr(ent, inst, period, tstActvTmr); /* Register timer function*/
```

5.3 Message Activation

The message activation function is called by system services to activate a task whenever another task has posted a message to it.

The message activation function is registered with system services by the layer management entity that is building the protocol stack on the system.

The message activation function must be registered before a task can be scheduled for any message event.

Sample of the code in the layer management entity to register the message activation function for **SSINT2**:

```
Ent      ent;
Inst     inst;
Priority priority;
ent = TSTENT;                      /* Entity ID of task */
inst = TSTINST0;                   /* Instance ID of task */
priority = PRIOR0;                 /* Priority of task */
SRegActvTsk(ent, inst, TTNORM, priority, tstActvTask);
Sample of the code in a task to create, build and post a message to
another task for SSINT2:
Pst    pst;
Buffer mBuf;
/* Get a message */
SGetMsg(pst.region, pst.pool, &mBuf);

/* Add 16 bytes of data to the end of the message */
for (i = 0; i < 16; i++)
  SAddPstMsg((Data) i, mBuf);

/* Post the message to the test task */
pst.event = (Event) EVTBNDREQ; /* event type */
SPstTsk(&pst, mBuf);
```

5.4 Memory Management

The memory management functions allocate and deallocate variable sized static and dynamic buffers.

Sample of the code in a task to allocate a static memory pool and then to allocate and deallocate static buffers within that pool:

```
Pool sPool;
Data *sBuf;
Region region;
region = TSTREG;

/* Get static memory pool of 100 bytes */
SGetSMem(region, (Size) 100, &sPool);

/* Get static buffer of 20 bytes from static memory */
SGetSBuf(region, sPool, &sBuf, (Size) 20);

/* Put static buffer of 20 bytes to static memory */
SPutSBuf(region, sPool, sBuf, (Size) 20);
```

5.5 Message Management

The message management functions initialize, add, and remove data to and from messages utilizing dynamic buffers.

Sample of the code in a task to initialize, add, examine, and remove data to and from a message:

```
Queue  q;
MsgLen msglen;
Buffer *m;
Data   data;
Region region;
Pool   pool;
region = TSTREG;
pool = TSTPOOL;

/* Get message */
SGetMsg(region, pool, &m);

/* Find length of message, value must be 0 */
SFndLenMsg(m, &msgLen);

/* Add byte to message */
SAddPreMsg((Data) 0xaa, m);

/* Find length of message, value must be 1 */
SFndLenMsg(m, &msgLen);
```

```
/* Add byte to message */
SAddPreMsg((Data) 0x55, m);

/* Find length of message, value must be 2 */
SFndLenMsg(m, &msgLen);

/* Examine message at index 0, value must be 0x55 */
SExamMsg(&data, m, (MsgLen) 0);

/* Examine message at index 1, value must be 0xaa */
SExamMsg(&data, m, (MsgLen) 1);

/* Remove byte from message, value must be 0x55 */
SRemPreMsg(&data, m);

/* Find length of message, value must be 1 */
SFndLenMsg(m, &msgLen);

/* Remove byte from message, value must be 0xaa */
SRemPreMsg(&data, m);

/* Find length of message, value must be 0 */
SFndLenMsg(m, &msgLen);

/* Put message */
SPutMsg(m);
```

5.6 Queue Management

The queue management functions initialize, add, and remove messages to and from queues.

Sample of the code in a task to initialize, add, and remove messages to and from a queue:

```
Queue  q;
QLen   qlen;
Buffer *m;

/* Task initializes queue */
SInitQueue(&q);

/* Find length of queue, value must be 0 */
SFndLenQueue(&q, &qlen);

/* Get message */
SGetMsg(region, pool, &m);

/* Add message to queue */
SQueueFirst(m, &q);

/* Find length of queue, value must be 1 */
SFndLenQueue(&q, &qlen);

/* Remove message from queue */
SDequeueFirst(&m, &q);

/* Find length of queue, value must be 0 */
SFndLenQueue(&q, &qlen);
```

6

Porting Issues

There are a number of factors that must be considered during the partition of the System Services Interface. Some of the factors are as follows.

6.1 Stack

Determine the protocol stack organization. Each protocol stack consists of one or more protocol layers. Each protocol layer can be considered as a separate task.

6.2 Processors

1. Determine the processor organization.
2. Determine which processors intercommunicates with each other.
3. For each processor that intercommunicates, determine if communications are through memory region or communications channel.
4. Specify the processor IDs.

6.3 Tasks

1. Determine the task organization. The location of each task depends on the data flow between tasks, processor performance, available memory, and access to other hardware resources.
2. Specify globally unique entity and instance IDs for each task. The entity typically represents a protocol layer and the instance typically represents a processor on which the entity resides.
3. Specify a memory region ID that a task may use for static memory during initialization.

4. Specify the memory region and memory pool IDs that a task may use to communicate with other tasks through its various interfaces.
5. Specify the priority and route that a task may use to communicate with other tasks through its various interfaces.
6. Specify the entity and instance that a task may use to communicate with other tasks through its various interfaces.
7. Determine which tasks intercommunicates through tightly or loosely coupled interfaces. Tightly coupled interfaces provide good performance, but have significant issues of scheduling fairness. Loosely coupled interfaces provide fair performance, and do not have significant issues of scheduling fairness.

6.4 Memory Regions

1. Determine which memory regions are shared and private.
2. Determine the start and end addresses (that is, size) for each memory region.
3. Determine which processors access each memory region.
4. Determine which processor is responsible for initializing the memory region.
5. Specify globally unique IDs for each memory region. These IDs are used in the initialization, memory management, message management, semaphores, and miscellaneous functions.

6.5 Memory Pools

1. Determine the memory pool organization within each memory region. This depends on processor organization, buffer organization, and the task configuration.
   - **Processor organization**: Determines whether one or more processors are accessing the memory pool.
   - **Buffer organization**: Determines whether a single or different pool is used to provide buffers for both control points and messages. Control points and messages have variable sizes. Control points are typically allocated and deallocated less frequently than messages.
   - **Task configuration**: Determines the maximum number and size of the messages that may be queued by the task. This is determined by the architecture and performance of the system. Also, determines the maximum number of control points and their sizes supported by the task.
2. Determine the start and end addresses (that is, size) for each pool.
3. Specify locally unique (within the memory region) IDs for each pool. These IDs are used in the initialization, memory management, message management, and miscellaneous functions.

6.6 Print/Display

1. Determine the print/display interface.
2. The print/display functions may be called only to provide debugging information.

6.7 Interrupts

1. Determine the interrupt organization.
2. Map **`SGetVect()`**, **`SPutVect()`**, **`SDisInt()`**, and **`SEnbInt()`** to the interrupt architecture of the processor.

6.8 Error

Determine the error reporting interface.

References

Refer to the following for additional information:

Coding Standards, Continuous Computing Corporation (p/n 1049002).

Documentation Standards, Continuous Computing Corporation (p/n 1049001).

Quality Plan, Continuous Computing Corporation (p/n 1050001).

Open Systems Interconnection - Basic Reference Model, ISO (IS 7498).

Open Systems Interconnection - Basic Reference Model Addendum 1: Connection-less Data Transmission, ISO (IS 7498 DAD 1).

Reference Model of Open Systems Interconnection for CCITT Applications, ITU-T(CCITT), (X.200).