SYSTEM SERVICES INTERFACE

Service Definition
1111001 1.6

SYSTEM SERVICES INTERFACE

Service Definition
1111001 1.6

Trillium Digital Systems, Inc.
12100 Wilshire Blvd., Suite 1800
Los Angeles, CA 90025-7118
Phone: +1 (310) 442-9222
Fax: +1 (310) 442-1162
Web: http://www.trillium.com

System Services Interface
Service Definition
1111001 1.6

Preface

Objective

This document provides a detailed description of the services provided by the System Services Interface software designed by Trillium Digital Systems, Inc.

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
Preface	This section gives information about the purpose and scope of the document, such as the suggested order of reading, the document set, and the font conventions used.
Contents	This is a detailed table of contents.
List of Illustrations	This is a list of all the figures in the document and the pages on which they appear.
1 Introduction	This section gives a textual and graphical overview of the product. It also contains Trillium-specific abbreviations.
2 Environment	This section describes assumptions about the software environment for operation of the System Services Interface software.
3 Interface Primitives	This section describes in detail the interface primitives at the System Services Interface layer interfaces.
Glossary	This section lists technical abbreviations and their expansions, and explains technical concepts and terms.
Reference	This section tells you about related product documents, technical specifications, and reports you can reference for additional information and understanding.
Index	This is an alphabetical index of key words and concepts. This supplements the table of contents for referencing and accessing the document.

Notations

This table displays the notations used in this document.

Notation	Explanation	Examples
Arial	**Titles**	**1.1 Title**
Palatino	Body text	This is body text.
Bold, *italics*	**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;`**

Release History

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

Version	Date	Initials	Description
1.6	10/16/97 03/24/98	aa ada	• Added F/Q System Service • Added SAddMsgRef System Service
1.5	08/06/96	fg	• Added new system service interface primitives • Updated environment descriptions

Contents

Preface 5

Illustrations 13

1 INTRODUCTION 15

2 ENVIRONMENT 17

2.1 Processors 17
2.2 Tasks and Activation Functions 17
2.2.1 Driver Tasks 17
2.2.2 Inter-Task Communication 18
2.2.3 Activation Function Types 18
2.2.3.1 Initialization Activation Function 18
2.2.3.2 Normal Message Activation Function 18
2.2.3.3 Timer Activation Function 18
2.2.3.4 Permanent Activation Function 18
2.2.4 Activation Function Scheduling 19
2.3 Memory Regions, Pools and Buffers 19
2.3.1 Memory Pool Types 19
2.3.1.1 Static Memory Pools 19
2.3.1.2 Dynamic Memory Pools 19
2.3.2 Memory Buffer Types 20
2.3.2.1 Static Memory Buffers 20
2.3.2.2 Dynamic Memory Buffers 20
2.4 Messages 20
2.4.1 Message Priority 21
2.4.2 Message Routing 21
2.4.3 Message Sharing 22
2.5 Queues 22

3 Trillium Advanced Portability Architecture (TAPA) 23

3.1 Layer Interfaces 23

3.1.1 Upper Layer Interface 24
3.1.2 Lower Layer Interface 24
3.1.3 Layer Management Interface 25
3.1.4 System Services Interface 25
3.1.5 Service Access Points (SAPs) 26
3.1.6 Layer Coupling 27
3.1.6.1 Tight Coupling 27
3.1.6.2 Loose Coupling 27
3.1.6.3 Interface Primitive Resolution 27
3.1.7 Multiple Service Users and Service Providers 31
3.1.8 Single or Multi-Memory Architectures 31
3.1.9 Single or Multitasking Architectures 32
3.1.10 Error Checking and Recovery 32
3.1.10.1 Types of Error Checks 32
3.1.10.1.1 Protocol Error Checks 32
3.1.10.1.2 Interface Error Checks 32
3.1.10.1.3 Input Checks 32
3.1.10.1.4 Output Checks 32
3.1.10.1.5 Resource Errors 33
3.1.10.1.6 Debug Errors 34
3.1.10.2 Granularity of Control Over Error Checks 34
3.1.10.2.1 Error Class Flags 35
3.1.10.3 Recovery Action Upon Error Detection 36
3.1.10.3.1 Generating Error Indications 36
3.1.10.3.2 Logging the Error With System Services 36
3.1.10.3.3 Generating Layer Manager Alarms 37
3.1.10.3.4 Error Logging vs. Alarms 37
3.1.10.3.5 Debug Printing 37
3.1.10.3.6 Returning to Stable State 38
3.1.10.3.7 Ignoring the Error 38
3.1.10.3.8 Truncating the Event Processing 38
3.1.10.3.9 Rolling Back the State 38

4 SYSTEM SERVICES 39

4.1 Initialization Functions 39
4.2 Timer Functions 39
4.3 Event Function 39
4.4 Driver Functions 40
4.5 Memory Management Functions 41
4.6 Message Functions 41
4.7 Queue Management Functions 42
4.8 Miscellaneous Functions 43
4.9 Multi-Threaded Primitives 43
4.10 Microsoft Windows NT Kernel Primitives 44
4.11 Type Definitions 45
4.12 Function Specifics 48

4.12.1 Initialization Functions 48
4.12.1.1 SRegInit 48
4.12.1.2 xxActvInit 51
4.12.2 Timer Functions 53
4.12.2.1 SRegTmr 53
4.12.2.2 SDeregTmr 55
4.12.2.3 xxActvTmr 56
4.12.3 Event Functions 57
4.12.3.1 SRegActvTsk 57
4.12.3.2 SDeregInitTskTmr 59
4.12.3.3 SPstTsk 60
4.12.3.4 xxActvTsk 61
4.12.3.5 SExitTsk 62
4.12.4 Driver-Related Scheduling Functions 63
4.12.4.1 SRegDrvrTsk 63
4.12.4.2 SAlignDBufEven 65
4.12.4.3 SChkMsg 66
4.12.4.4 SSetIntPend 67
4.12.4.5 SExitInt 68
4.12.4.6 SEnbInt 69
4.12.4.7 SDisInt 70
4.12.4.8 SHoldInt 71
4.12.4.9 SRelInt 72
4.12.4.10 SGetVect 73
4.12.4.11 SPutVect 74
4.12.4.12 SGetEntInst 75
4.12.4.13 SSetEntInst 76
4.12.4.14 SGetDBuf 77
4.12.4.15 SPutDBuf 78
4.12.4.16 SAddDBufPst 79
4.12.4.17 SAddDBufPre 80
4.12.4.18 SRemDBufPst 81
4.12.4.19 SRemDBufPre 82
4.12.4.20 SGetDataRx 83
4.12.4.21 SGetDataTx 84
4.12.4.22 SInitNxtDBuf 85
4.12.4.23 SGetNxtDBuf 86
4.12.4.24 SChkNxtDBuf 87
4.12.4.25 SUpdMsg 88
4.12.5 Memory Management Functions 89
4.12.5.1 SGetSMem 89
4.12.5.2 SPutSMem 91
4.12.5.3 SGetSBuf 92
4.12.5.4 SPutSBuf 93
4.12.6 Message Functions 94
4.12.6.1 SGetMsg 94
4.12.6.2 SPutMsg 95
4.12.6.3 SInitMsg 96
4.12.6.4 SFndLenMsg 97
4.12.6.5 SExamMsg 98

4.12.6.6 SRepMsg 99
4.12.6.7 SAddPreMsg 100
4.12.6.8 SAddPstMsg 101
4.12.6.9 SRemPreMsg 102
4.12.6.10 SRemPstMsg 103
4.12.6.11 SAddPreMsgMult 104
4.12.6.12 SAddPstMsgMult 105
4.12.6.13 SGetPstMsgMult 106
4.12.6.14 SRemPreMsgMult 107
4.12.6.15 SRemPstMsgMult 109
4.12.6.16 SCpyFixMsg 111
4.12.6.17 SCpyMsgFix 113
4.12.6.18 SCpyMsgMsg 115
4.12.6.19 SCatMsg 117
4.12.6.20 SSegMsg 119
4.12.6.21 SCompressMsg 121
4.12.6.22 SAddMsgRef 122
4.12.6.23 SPkS8 124
4.12.6.24 SPkU8 125
4.12.6.25 SPkS16 126
4.12.6.26 SPkU16 127
4.12.6.27 SPkS32 128
4.12.6.28 SPkU32 129
4.12.6.29 SUnpkS8 130
4.12.6.30 SUnpkU8 131
4.12.6.31 SUnpkS16 132
4.12.6.32 SUnpkU16 133
4.12.6.33 SUnpkS32 134
4.12.6.34 SUnpkU32 135
4.12.7 Queue Management Functions 136
4.12.7.1 SInitQueue 136
4.12.7.2 SQueueFirst 137
4.12.7.3 SQueueLast 138
4.12.7.4 SDequeueFirst 139
4.12.7.5 SDequeueLast 140
4.12.7.6 SFlushQueue 141
4.12.7.7 SCatQueue 142
4.12.7.8 SFndLenQueue 144
4.12.7.9 SAddQueue 145
4.12.7.10 SRemQueue 146
4.12.7.11 SExamQueue 147
4.12.8 Miscellaneous Functions 148
4.12.8.1 SFndProcId 148
4.12.8.2 SSetProcId 149
4.12.8.3 SSetDateTime 150
4.12.8.4 SGetDateTime 152
4.12.8.5 SGetSysTime 154

4.12.8.6 SRandom 155
4.12.8.7 SError 156
4.12.8.8 SLogError 157
4.12.8.9 SChkRes 159
4.12.8.10 SPrint 160
4.12.8.11 SDisplay 161
4.12.8.12 SPrntMsg 162
4.12.9 Multi-threaded System Service Primitives 163
4.12.9.1 SGetMutex 163
4.12.9.2 SPutMutex 164
4.12.9.3 SLockMutex 164
4.12.9.4 SUnlockMutex 166
4.12.9.5 SGetCond 167
4.12.9.6 SPutCond 168
4.12.9.7 SCondWait 169
4.12.9.8 SCondSignal 171
4.12.9.9 SCondBroadcast 172
4.12.9.10 SGetThread 173
4.12.9.11 SPutThread 175
4.12.9.12 SThreadYield 176
4.12.9.13 SThreadExit 177
4.12.9.14 SSetThrdPrior 178
4.12.9.15 SGetThrdPrior 179
4.12.9.16 SExit 180
4.12.10 Microsoft Windows NT Kernel Primitives 181
4.12.10.1 SPutIsrDpr 181
4.12.10.2 SSyncInt 182

5 USAGE 183

5.1 Initialization 183
5.2 Timer Activation 183
5.3 Message Activation 184
5.4 Memory Management 185
5.5 Message Management 186
5.6 Queue Management 188

6 PORTATION ISSUES 189

6.1 Stack 189
6.2 Processors 189
6.3 Tasks 189
6.4 Memory Regions 189

6.5 Memory Pools 190
6.6 Print/Display 190
6.7 Interrupts 190
6.8 Error 190

7 ABBREVIATIONS 191

8 REFERENCES 193

Illustrations

Figure 3-1: TAPA 23
Figure 3-2: Service Access Points (SAPs) 26
Figure 3-3: Operations for primitive resolution: loosely coupled interface 28
Figure 4-1: Data flow: initialization function 49
Figure 4-2: Data flow: timer function 54
Figure 4-3: Data flow: event function 58

1 INTRODUCTION

This document provides the definition of the system service interface used by all software products designed by Trillium Digital Systems, Inc. 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 should typically be read in conjunction with the Service Definition of a particular product.

2 ENVIRONMENT

This section describes the assumptions about the environment in which the software product was 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 or are shared with other processors.

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

2.2 Tasks and Activation Functions

A task is an invocation of a Trillium product.

An activation function is a schedulable 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 (e.g., protocol type). Common entity ids are defined in **ssi.h**. 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 (e.g., two copies of a protocol layer).

2.2.1 Driver Tasks

The system service interface provides for the concept of "Driver Tasks." In this discussion, driver tasks relate to special tasks registered with system services (see **SRegDrvrTsk**) that provide communications links for moving messages (via loosely coupled interfaces) between separate stack implementations that cross arbitrary boundaries (i.e., 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 via direct function calls (tight coupling) or via 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 via 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 2.2.3.1-2.2.3.4, below.

Activation function types are defined in **ssi.h**. 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 via 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.

2.2.4 Activation Function Scheduling

Activation functions are scheduled depending on their type--for example, timer functions are scheduled at prespecified 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 described below.

2.3.1.1 Static Memory Pools

"Static" pools are used for allocating static buffers (described below). 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 (see **SGetSMem()**).

2.3.1.2 Dynamic Memory Pools

"Dynamic" pools are used for allocating dynamic buffers (described below) 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 a later section of this document. 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 described below.

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 the next section). 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 could 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 will be 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`**.

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 will need 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**:

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 (via loosely coupled interfaces). How priorities are used depends on the system service 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 occurs 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**.:

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 (via loosely coupled interfaces). How routes are used depends on the system service interface implementation.

2.4.3 Message Sharing

A message may be shared by two or more users to try and avoid message duplication and 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 would indicate 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 should 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 will be 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`**).

Queues may be implemented as a contiguous message pointer array accessed via 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.

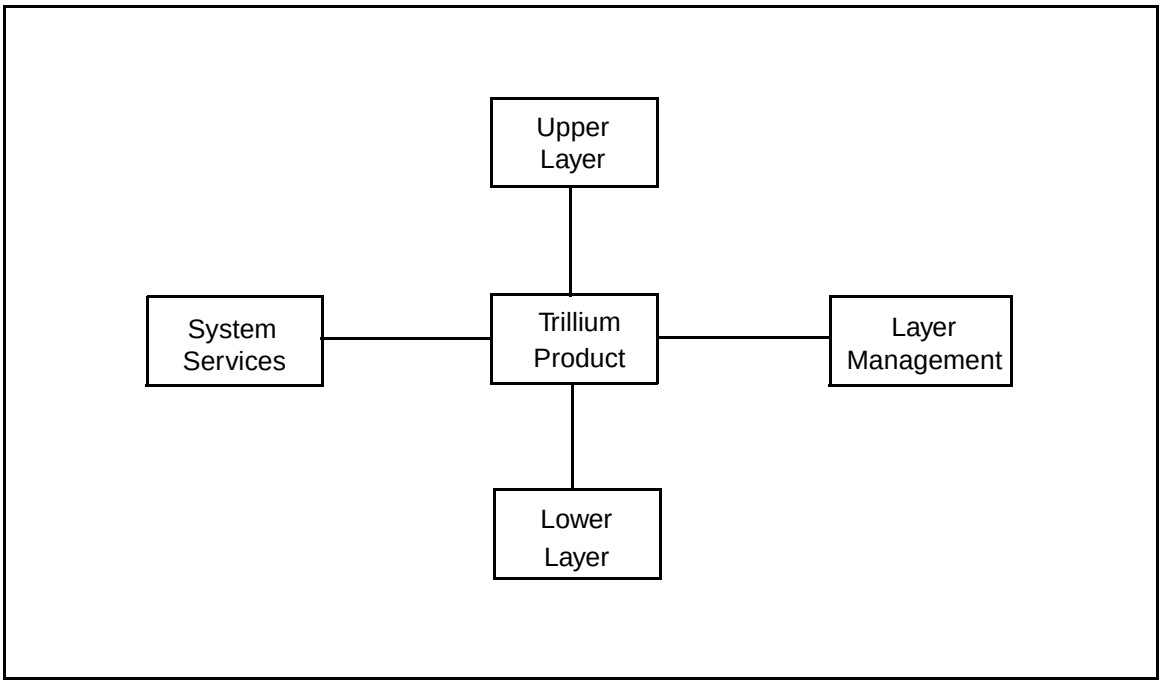

Figure 3-1: TAPA

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:

Function	Description
Bind/Unbind	Registers and deregisters service user with service provider

Function	Description
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) has been 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:

Function	Description
Bind/Unbind	Registers and deregisters 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) has been 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:

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 purposes
Tracing	Logs protocol messages to the layer manager for tracing purposes
Control	Activates and deactivate 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

The system service interface provides the operating system functions required by the Trillium product. These functions are:

Function	Description
Initialization	Initializes the protocol layer
Timer Management	Provides for the periodic activation of the protocol layer
Task Scheduling	Registers, deregisters, 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 service interface is the same for each product. Not all system service 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, etc.). 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 illustrated in the following diagram:

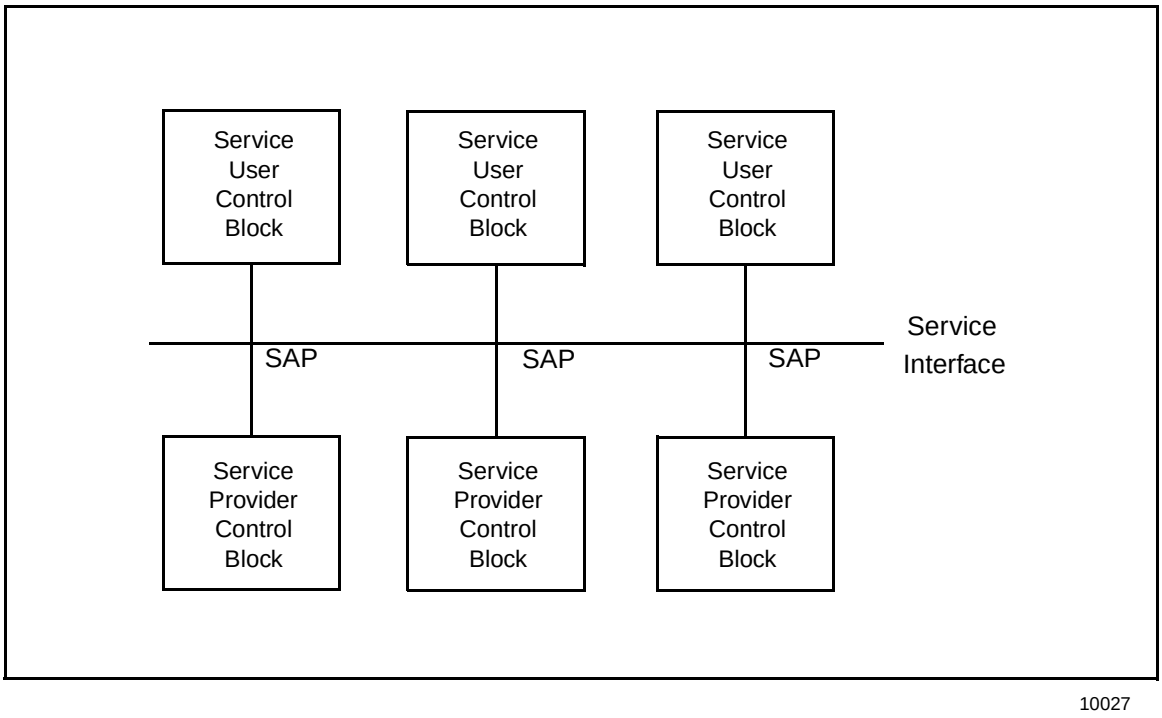

Figure 3-2: Service Access Points (SAPs)

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 via direct function calls (tight coupling) or via 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 via 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 will do so via 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 following flow diagram:

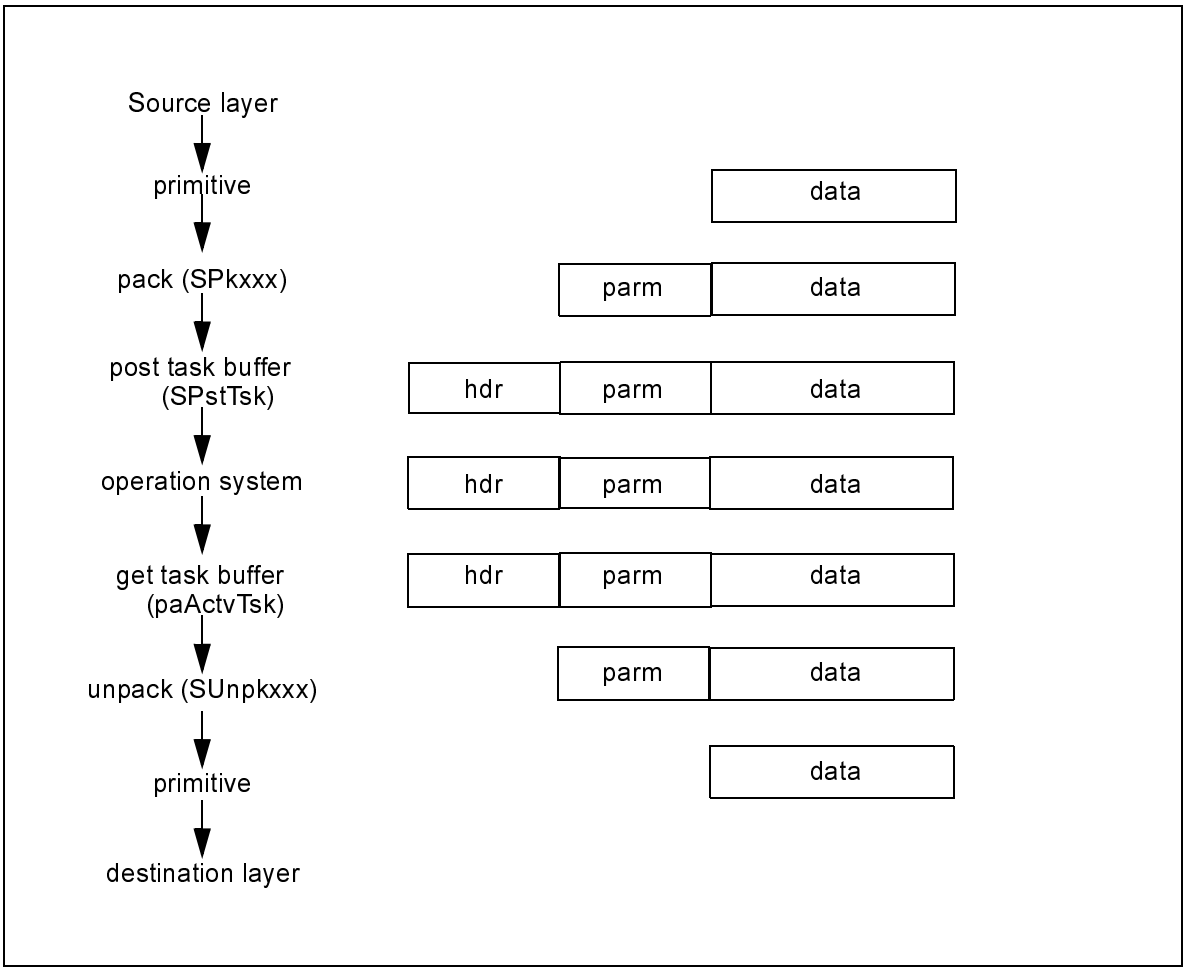

Figure 3-3: Operations for primitive resolution: loosely coupled interface

The following example for the **EcLiAmtBndReq** primitive illustrates 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, via 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, via 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 described below.

3.1.10.1.1 Protocol Error Checks

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

- Receiving an unexpected or improper protocol message
- Timer expiry

Such checks will always be 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. Examples:

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

There are two classes of error checks here, as described below.

3.1.10.1.3 Input Checks

These checks relate to validating the parameters supplied in an interface primitive or common function (e.g., checking if the **`spId`** supplied in a **`BndReq`** primitive is for a configured SAP). Such checks may be disabled once the interfaces have been 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`** was successful). Output errors from system services and common functions shall generally be checked for.

3.1.10.1.5 Resource Errors

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

- 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 will always be 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 will always be 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 will succeed. 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. Examples:

- 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 would not happen were the codes 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 has been designed (and tested) to never encounter those errors. To this end, certain error class flags **(ERRCLS_*)** are defined that should surround the corresponding types of error checks, so they can be disabled at compile time if so desired.

3.1.10.2.1 Error Class Flags

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

/* We are 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 will disable 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 will be 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 in more detail below.

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 in more detail now.

3.1.10.3.2 Logging the Error With System Services

This is done via the system service primitive, **SLogError**, described later in this document.

3.1.10.3.3 Generating Layer Manager Alarms

Layer manager alarms (status indications) are generated to inform the layer manager about such things as:

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

In general, alarms will be generated to report errors that indicate a system design or configuration problem. Normally, the layer manager will not be 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 vs. 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 will be logged via **`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 will be 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 typically 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 Processing

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--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 need to be returned to the pool.

4 SYSTEM SERVICES

This section describes all functions defined at the system service 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

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

4.2 Timer Functions

Name	Description
SRegTmr	Register activation function - timer
SDeRegTmr	Deregister activation function - timer
xxActvTmr	Timer activation function for a task

4.3 Event Function

Name	Description
SRegActvTsk	Register message handling function
SDeregInitTskTmr	Deregister 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

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
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:

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:

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

Name	Description
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
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:

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:

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 service 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 will be 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.

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

Name	Description
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
SExit	Exit system service process gracefully (cleanup)

4.10 Microsoft Windows NT Kernel Primitives

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

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

4.11 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 typedefs 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);
```

4.12 Function Specifics

The following section describes the specific calling parameters and sequence for the system service 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 its allowable values

4.12.1 Initialization Functions

4.12.1.1 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 will invoke the function passed to it once before scheduling the task with any other events. Typically, it will schedule the function immediately. The initialization function will be used by the task to initialize its global variables.

The data flow is:

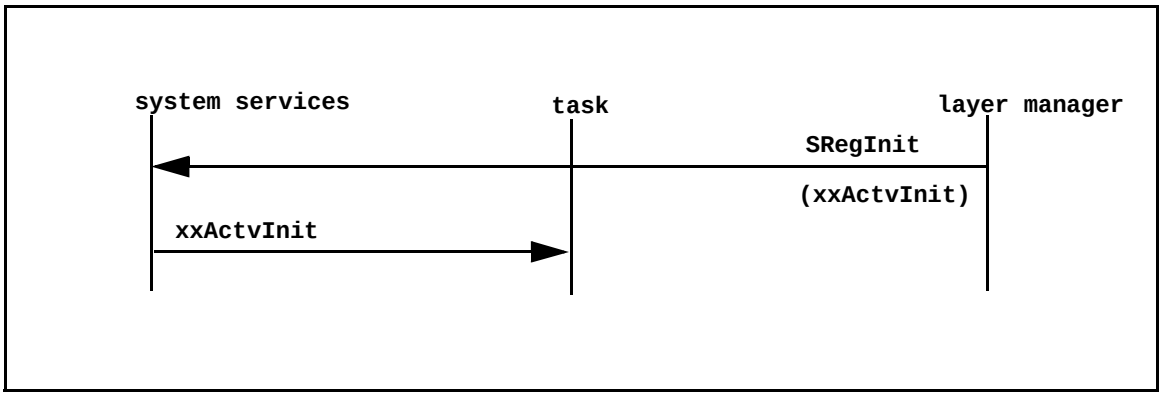

Figure 4-1: Data flow: initialization function

Returns:

ROK OK

RFAILED failed: illegal parameters

4.12.1.2 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 will invoke this function, passed to it via **SRegInit**, exactly once, before scheduling the task with any other events. The task will use the initialization function to initialize its global variables.

Returns:

ROK OK

RFAILED failed: illegal parameters

4.12.2 Timer Functions

4.12.2.1 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 will periodically invoke the function passed to it at the specified intervals. The timer function will be used by the task to manage its internal timers (e.g., protocol timers). A .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:

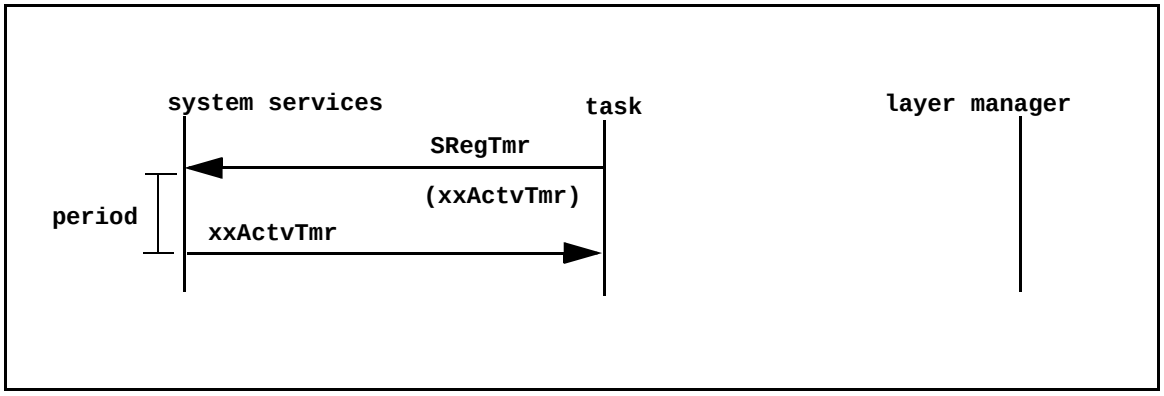

Figure 4-2: Data flow: timer function

Returns:

ROK OK

RFAILED failed: illegal parameters

4.12.2.2 SDeregTmr

Name:

Deregister 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 deregistering the timer. Allowable values: 0 - 255

`inst`

Instance of task deregistering 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 deregister a timer function. The timer resource is deleted from the system.

Returns:

`ROK` OK

`RFAILED` failed: illegal parameters

4.12.2.3 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 will be used by the task to manage its internal timers (e.g., protocol timers).

Returns:

ROK OK

RFAILED failed: illegal parameters

4.12.3 Event Functions

4.12.3.1 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 will schedule the activation function passed to it whenever a message is received that is destined for this task. The task will use the activation function to receive and handle messages from other tasks.

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

The data flow is:

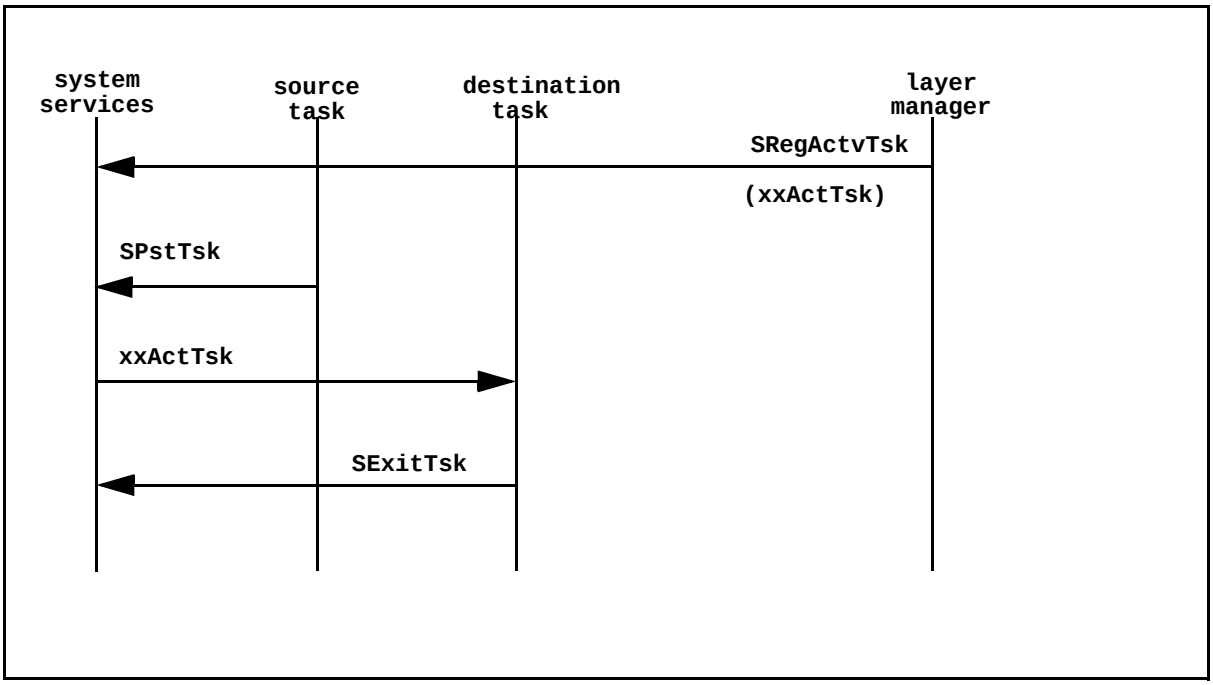

Figure 4-3: Data flow: event function

Returns:

ROK OK

RFAILED failed: illegal parameters.

4.12.3.2 SDeregInitTskTmr

Name:

Deregister 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 deregister the initialization, message, and timer activation functions for a task.

Returns:

ROK OK

RFAILED failed: illegal parameters, unable to free memory

4.12.3.3 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, etc.

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

4.12.3.4 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, etc.

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 have been registered earlier using **SRegActvTsk**. It must exit with a call to **SExitTsk** under the preemptive scheduler.

Returns:

ROK OK

RFAILED failed: illegal parameters

4.12.3.5 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 **xxActvTmr** functions) must exit with a call to this function. This routine allows "task conscious" schedulers to monitor task activations.

Returns:

ROK OK

4.12.4 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).

4.12.4.1 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 should 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 should call when its interrupt pending flag has been set via 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 should be passed to the activation function for delivery. Upon receipt of an interrupt, the driver entity should notify system services of a pending interrupt via the **`SSetIntPend()`** system service routine. As soon as possible (asynchronously), system services should then schedule the **`isTsk`** to process the interrupt.

Returns:

`ROK` OK

`RFAILED` failed: illegal parameters

4.12.4.2 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

4.12.4.3 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

4.12.4.4 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/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 via this system service routine. As soon as possible (asynchronously), system services should then schedule the appropriate registered interrupt service task to process the interrupt.

Returns:

`ROK` OK

`RFAILED` failed

4.12.4.5 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

4.12.4.6 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

4.12.4.7 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

4.12.4.8 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 should 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

4.12.4.9 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 OS or layer1 hardware drivers.

Returns:

ROK OK

RFAILED failed

4.12.4.10 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

4.12.4.11 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

4.12.4.12 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 will be placed.

inst

Pointer to location where instance id will be 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

4.12.4.13 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

4.12.4.14 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

4.12.4.15 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

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

4.12.4.16 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

Message buffer

dBuf

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

4.12.4.17 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

Message buffer

dBuf

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

4.12.4.18 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

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

4.12.4.19 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

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

4.12.4.20 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

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 would be the maximum number of data bytes that can be placed in the data buffer.

Returns:

ROK OK

RFAILED failed: error

4.12.4.21 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

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 would be the actual number of data bytes in the data buffer.

Returns:

ROK OK

RFAILED failed: error

4.12.4.22 SInitNxtDBuf

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

Message

Description:

This function initializes the next data buffer Id in the message to point to the first data buffer in the message. It should 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

4.12.4.23 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

Message

dBuf

Data buffer to be returned

Description:

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

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.4.24 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

Message

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

4.12.4.25 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

Message

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

4.12.5 Memory Management Functions

4.12.5.1 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 has been configured by the layer manager with the maximums it will need to support.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.5.2 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

4.12.5.3 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 will be 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

4.12.5.4 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

4.12.6 Message Functions

4.12.6.1 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 - 255

mBufPtr

Pointer to the location where the allocated message pointer will be 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, etc.

4.12.6.2 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 the message

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, etc.

4.12.6.3 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 the message

Description:

This function reinitializes 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, etc.

4.12.6.4 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 the message

lngPtr

Pointer to the location where the length of the message will be 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, etc.

4.12.6.5 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 will be placed

mBuf

Pointer to the message

idx

Index into message (zero-based) where data will read 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, etc.

4.12.6.6 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 will replace existing data in the message

mBuf

Pointer to the message

idx

Index into message (zero-based) where data will be 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, etc.

4.12.6.7 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 the message

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

4.12.6.8 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

4.12.6.9 SRemPreMsg

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 will be placed

mBuf

Pointer to the message

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

4.12.6.10 SRemPstMsg

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 will be placed

mBuf

Pointer to the message

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

4.12.6.11 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

Pointer to the message

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 will become the first byte of the message. For example, if a data string "1234" is added to the message "xyz," the new message contents will be "4321xyz."*

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.12 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

Pointer to the message

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 will become the last byte of the message. For example, if a data string "1234" is added to the message "xyz," the new message contents will be "xyz1234."*

Returns:

ROK OK

RFAILED failed: error

4.12.6.13 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

4.12.6.14 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 will be 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 will become the first byte of the data string. For example, if 4 bytes are removed from the message "1234xyz," the new message will be "xyz" 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

4.12.6.15 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 will be 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 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 databytes 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 will become the first byte of the data string. For example, if 4 bytes are removed from the message "xyz1234," the new message will be "xyz" 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

4.12.6.16 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 will be placed.

Description:

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 will become the first byte of the message. For example, if a data string "1234" is added to the message "xyz" at index 0, the new message contents will be "4321xyz."*

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.17 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 will be placed.

cCnt

Pointer to location where the count of number of bytes of data actually copied from the message will be 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

4.12.6.18 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 will be 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

4.12.6.19 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 1

mBuf2

Pointer to message 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

4.12.6.20 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 one - original message to be segmented.

idx

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

mBuf2

Pointer to message buffer two (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

4.12.6.21 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 the message

Description:

This function will compress 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

4.12.6.22 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 will be 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, should 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 should 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

4.12.6.23 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.24 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.25 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.26 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.27 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.28 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

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** should be used to unpack the data from a message packed by this function.

Returns:

ROK OK

ROUTRES failed: out of resources

RFAILED failed: error

4.12.6.29 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 will be placed.

mBuf

Pointer to message

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** should be used to pack the data to a message unpacked by this function.

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.6.30 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 will be placed.

mBuf

Pointer to message

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** should be used to pack the data to a message unpacked by this function.

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.6.31 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 will be placed

mBuf

Pointer to message

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** should be used to pack the data to a message unpacked by this function.

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.6.32 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 will be placed

`mBuf`

Pointer to message

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`** should be used to pack the data to a message unpacked by this function.

Returns:

`ROK` OK

`ROKDNA` OK: data not available

`RFAILED` failed: error

4.12.6.33 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 will be placed

mBuf

Pointer to message

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** should be used to pack the data to a message unpacked by this function.

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.6.34 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 will be placed

mBuf

Pointer to message

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** should be used to pack the data to a message unpacked by this function.

Returns:

ROK OK

ROKDNA OK: data not available

RFAILED failed: error

4.12.7 Queue Management Functions

4.12.7.1 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

4.12.7.2 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

4.12.7.3 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

4.12.7.4 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 dequeued message will be placed

q

Pointer to the queue

Description:

This function dequeues 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

4.12.7.5 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 dequeued message will be placed

q

Pointer to the queue

Description:

This function dequeues 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

4.12.7.6 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 dequeues 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 dequeued buffer is a message buffer, all data buffers associated with the message buffer are returned to memory.

Returns:

ROK OK

RFAILED failed: error

4.12.7.7 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 will have "b3, b2, b1, a1, a2, a3."*

Returns:

ROK OK

RFAILED failed: error

4.12.7.8 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 will be placed.

Description:

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

Returns:

ROK OK

RFAILED failed: error

4.12.7.9 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 will be placed in the queue.

Description:

This function inserts the message into the queue at the specified index. The index is zero- based (i.e., 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

4.12.7.10 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 dequeued message will be placed

q

Pointer to the queue

idx

Index into queue (zero-based) where queue will be examined

Description:

This function removes a message from the queue at the specified index. The index is zero- based (i.e., 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

4.12.7.11 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 dequeued message will be placed

q

Pointer to the queue

idx

Index into queue (zero-based) where queue will be examined

Description:

This function retrieves a message from the queue at the specified index. Index is zero-based (i.e., 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

4.12.8 Miscellaneous Functions

4.12.8.1 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

4.12.8.2 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

4.12.8.3 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

4.12.8.4 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 will be 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

4.12.8.5 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 will be 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

4.12.8.6 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 will be placed.

Description:

This function generates a pseudo random number. Refer to the article "Generating and Testing Pseudorandom Numbers" by Charles Whitney 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

4.12.8.7 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 should never return.

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

Returns:

RFAILED failed

4.12.8.8 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" (see section on error checking).

`0x2` **`ERRCLS_INT_PAR`** - error class "interface parameter" (see section on error checking).

`0x1` **`ERRCLS_DEBUG`** - error class "debug" (see 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`**. See the section on Error Checking and Recovery for more information of the use of this primitive.

Returns:

`RFAILED` failed

4.12.8.9 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 will be 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, etc.

Description:

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

Returns:

ROK OK

RFAILED failed: error

4.12.8.10 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 will be replaced by **SDisplay**.

Returns:

ROK OK

RFAILED failed: error

4.12.8.11 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 will replace **SPrint**.

Returns:

ROK OK

RFAILED failed: error

4.12.8.12 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

4.12.9 Multi-threaded System Service Primitives

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

4.12.9.1 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

4.12.9.2 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

4.12.9.3 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

4.12.9.4 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

4.12.9.5 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

4.12.9.6 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

4.12.9.7 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 (see **SCondSignal** and **SCondBroadcast**). Upon receipt of a signal, **SCondWait** reacquires the lock on the specified mutex and returns.

Example:

```
if (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

4.12.9.8 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

4.12.9.9 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

4.12.9.10 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 this 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: pass argument 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

4.12.9.11 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

4.12.9.12 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

4.12.9.13 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

4.12.9.14 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

4.12.9.15 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

4.12.9.16 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

4.12.10 Microsoft Windows NT Kernel Primitives

4.12.10.1 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

4.12.10.2 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

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, etc.).

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 has been 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 should be 0 */
SFndLenMsg(m, &msgLen);

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

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

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

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

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

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

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

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

```
/* remove byte from message, value should be 0xaa */
SRemPreMsg(&data, m);

/* find length of message, value should 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 should be 0 */
SFndLenQueue(&q, &qlen);

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

/* add message to queue */
SQueueFirst(m, &q);

/* find length of queue, value should be 1 */
SFndLenQueue(&q, &qlen);

/* remove message from queue */
SDequeueFirst(&m, &q);

/* find length of queue, value should be 0 */
SFndLenQueue(&q, &qlen);
```

6 PORTATION ISSUES

There are a number of factors that must be considered during the portation of the system service interface. Some of these factors are now listed.

6.1 Stack

Determine the protocol stack organization. Each protocol stack will consist of one or more protocol layers. Each protocol layer can be considered a separate task.

6.2 Processors

1. Determine the processor organization.
2. Determine which processors will intercommunicate with each other.
3. For each processor that will intercommunicate, determine if communications will be via a 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 via its various interfaces.
5. Specify the priority and route that a task may use to communicate with other tasks via its various interfaces.
6. Specify the entity and instance that a task may use to communicate with other tasks via its various interfaces.
7. Determine which tasks will intercommunicate via 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 will be shared and private.
2. Determine the start and end addresses (i.e., size) for each memory region.
3. Determine which processors access each memory region.
4. Determine which processor will be responsible for initializing the memory region.
5. Specify globally unique ids for each memory region. These ids will be 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: Determine whether one or more processors will be accessing the memory pool.
   - Buffer organization: Determine whether a single or different pool will be 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: Determine 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. Determine the maximum number of control points and their sizes supported by the task.
2. Determine the start and end addresses (i.e., size) for each pool.
3. Specify locally unique (within the memory region) ids for each pool. These ids will be used in the initialization, memory management, message management, and miscellaneous functions.

6.6 Print/Display

Determine the print/display interface.

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.

7 ABBREVIATIONS

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

8 REFERENCES

Refer to the following for additional information:

Coding Standards, Trillium Digital Systems, Inc. (p/n 1049002).

Documentation Standards, Trillium Digital Systems, Inc. (p/n 1049001).

Open Systems Interconnection - Basic Reference Model, ISO (IS 7498).

Open Systems Interconnection - Basic Reference Model Addendum 1: Connectionless Data Transmission, ISO (IS 7498 DAD 1).

Quality Plan, Trillium Digital Systems, Inc. (p/n 1050001).

Reference Model of Open Systems Interconnection for CCITT Applications, CCITT, (X.200).