TAPA Concepts and Guidelines

Common Document

1111011 1.11a

TAPA Concepts and Guidelines

Common Document

1111011 1.11a

Radisys Corporation

9450 Carroll Park Drive San Diego, CA 92121-2256

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

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

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

TAPA Concepts and Guidelines
Common Document
1111011 1.11a

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

Printed in U.S.A.

Contents

Figures

Tables

Preface

Objective

This document provides an overview of the TAPA Concepts and Guidelines followed by Continuous Computing Corporation.

Audience

The readers of this document are Continuous Computing customers who are keen to know about the TAPA environment and guidelines to write their application which interacts with the Trillium protocol stacks.

Document Organization

This document is referred to as TAPA and Trillium Protocol Layer or Protocol Layer in the rest of the document. It is organized into the following sections:

Section	Description
1 TAPA Environment	Describes the software environment in which Protocol Layer is designed to operate.
2 Writing Applications for Trillium's Portable Layer	Aspects to be considered when writing applications for the Protocol Layer.

Notations

This table displays the notations used in this document:

Notation	Explanation	Examples
Arial	**Titles**	**1.1 Title**
Book Antiqua	Body text	This is body text.
Bold	**Highlights information**	**Loose coupling, tight coupling, upper layer interface**
ALL CAPS	CONDITIONS, MESSAGES	AND, OR CONNECT ACK
Italics	*Document names, emphasis*	*TAPA Concepts and Guidelines Common Document.* This adds *emphasis*.
Courier New Bold	**Code Filenames, pathnames**	**PUBLIC S16 XxYyIntCfgReq (pst, cfg) Pst *pst; XxMngmt *cfg;**

Abbreviations

The following table defines the abbreviations used in this document.

Abbreviation	Description
CCITT	Consultative Committee for International Telegraph and Telephone. An international organization that develops communication standards, such as Recommendation X.25.
ISO	International Standards Organization.
ITU-T	International Telecommunications Union – Telecommunication Standardization Sector (formerly CCITT). This is also known as ITU-TSS.
OSI	Open Systems Interconnection.
PDU	Protocol Data Unit. A generic term for the format used to send information in a communications protocol, typically a packet with its headers and trailers.
SAP	Service Access Point.
TAPA	Trillium Advanced Portability Architecture.

For a general list of networking terms please refer to Continuous Computing's Online Glossary at http://www.ccpu.com/search/glossary/

Release History

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

Version	Date	Author (s)	Description
1.11a	October 10, 2011	Naveen Dcruz H	Addendum release for Radisys logo and template upgrade.
1.1	April 07, 2009	Srinivas Rao V.	Initial release.

1

TAPA Environment

Trillium Advanced Portability Architecture (TAPA) is a set of architectural standards to ensure that all Trillium products meet the objectives of:

- Portability
- Ease of Maintenance
- High Quality
- Consistency with other Trillium Products

This section describes the software environment in which the Trillium protocols are designed to operate.

1.1 Layer Interfaces

Figure 1-1 shows the Protocol Layer interfaces.

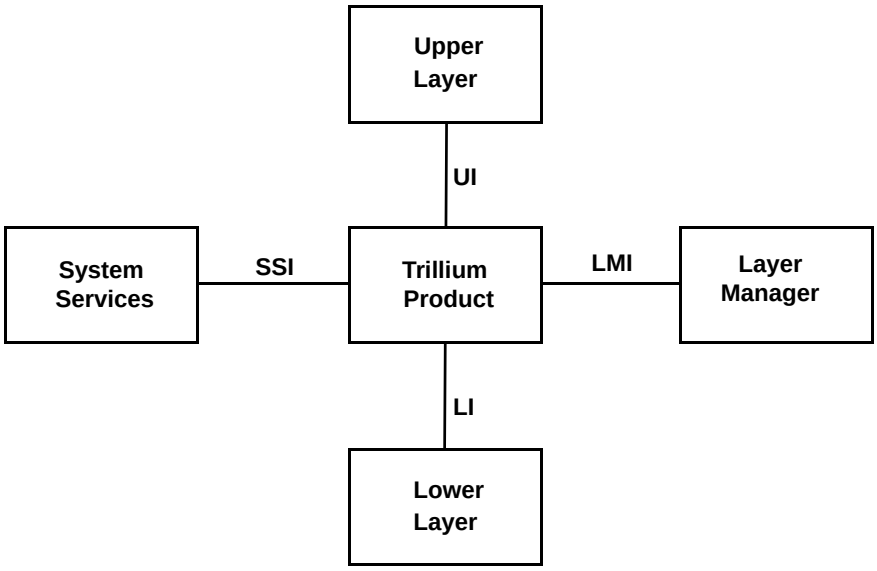

Figure 1-1: Trillium Advanced Portability Architecture

The following table lists the interfaces and describes their functions.

Table 1-1: TAPA Interfaces

Interface	Description
System Services Interface (SSI)	Provides functions such as buffer management, timer management, date/time management, resource checking, initialization. Refer to the *System Services Interface Service Definition* for details.
Layer Manager Interface (LMI)	Provides the necessary functions to configure, control, and monitor the condition of the Protocol Layer.
Upper Interface (UI)	The Upper Interface is used by the service user to access the Protocol Layer's services.
Lower Interface (LI)	The Lower Interface is used by the Protocol Layer to access it's services provider's services.

Protocol Layer interacts with the other layers and the layer manager by using the primitives and Service Access Points (SAPs) that are described later**.** Protocol Layer also interacts with system services by using a simple function interface**.**

Protocol Layer can be tightly or loosely coupled to other layers. For more details, refer to Section 2.1.2, "Coupling Options".

1.2 Service Access Points

Continuous Computing's TAPA adheres to the Open Systems Interconnection (OSI) reference model. A service user and service provider are present at an interface between two layers in a protocol stack. The service user accesses the services of the service provider at a Service Access Point (SAP). Both the service user and service provider have SAP control blocks that store the state of the SAPs and the SAP IDs. A one-to-one mapping, which defines the SAP, exists between a service user SAP control block and service provider SAP control block.

The Protocol Layer provides its services to the user through the Upper Interface SAP, and receives its services from the provider through the Lower Interface SAP. The exchange of the necessary primitives across the SAP provides a service to the user or receives a service from the provider.

Figure 1-2 shows the relationship between Protocol Layer and the adjacent layers.

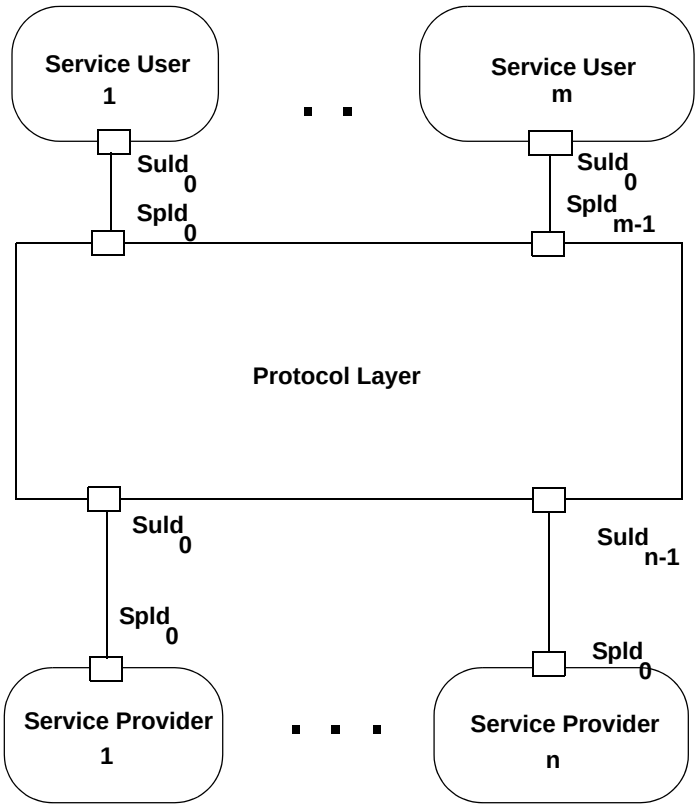

Figure 1-2: Layer Definition

The standardized interface of primitives and SAPs allows layers to be defined independent of each other. The peer-to-peer protocol of one layer does not affect any of the upper or lower layer protocols when modified, if layer interface requirements are met.

There is an implicit SAP between Protocol Layer and the layer manager. There is exactly one SAP at this interface, although it is not explicitly initialized or referenced. At this SAP, the layer manager configures and controls the Protocol Layer, retrieves statistics and status information for the Protocol Layer, and receives trace and alarm information from the Protocol Layer.

1.3 Primitives

The service user and service provider interact with each other using a set of primitive functions across a SAP.

Interaction between the Protocol Layer software and the upper layer, the lower layer, and the layer manager takes place using a set of primitive functions**.** The primitives either initiate, or are the result of, the interactions between two layers. Primitives completely define the interaction between layers, and take the form of:

- Requests (service user to service provider).
- Indications (service provider to service user).
- Responses (service user to service provider).
- Confirms (service provider to service user).

Figure 1-3 shows how primitives function across a SAP.

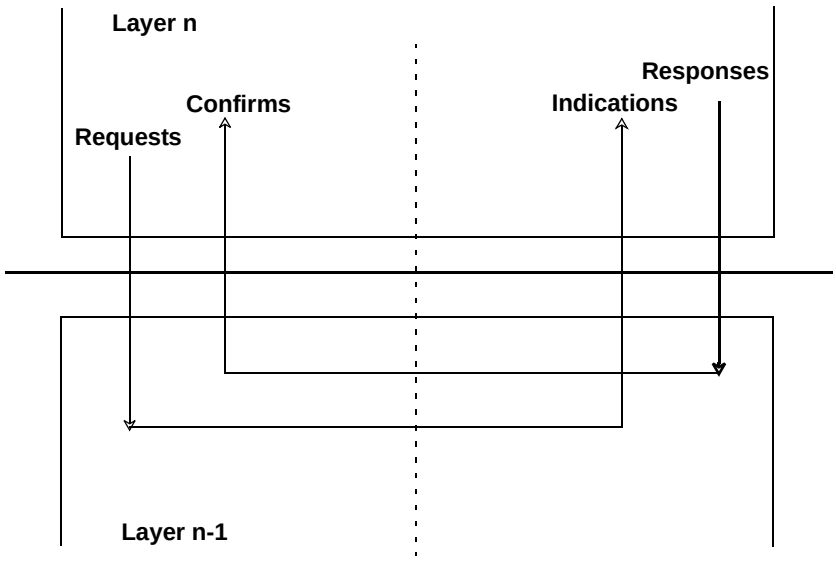

Figure 1-3: Primitive Functions

1.4 Primitive Names

Figure 1-4 shows how the interface primitive names are derived.

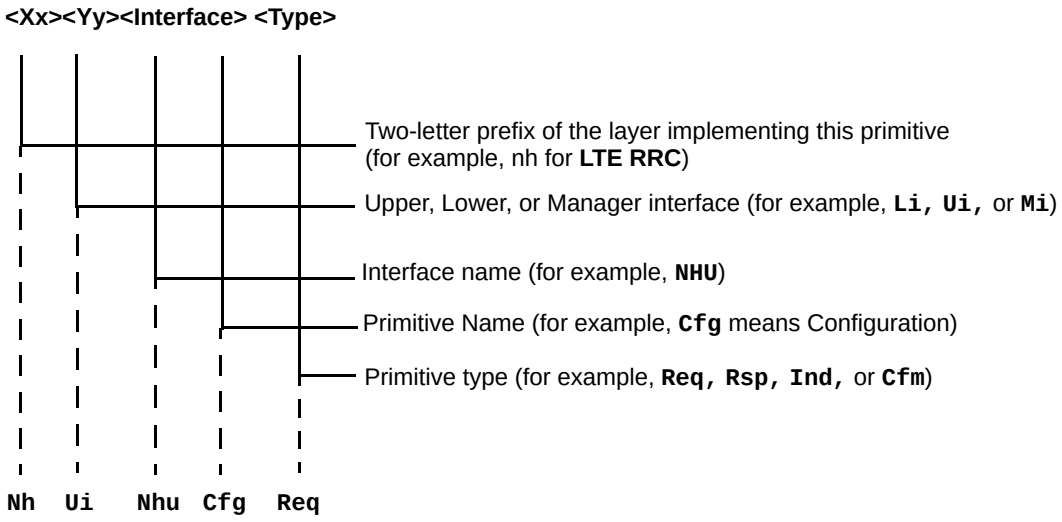

Figure 1-4: Primitive Naming Conventions

1.4.1 Naming Conventions

The following naming conventions apply to interface primitives.

- Primitive (interface) functions begin with an uppercase letter; the remaining characters are mixed case.
- **Xx** represents the layer name. **XxUiyyy**, **XxLiyyy**, and **XxMiLxx** represent the upper, lower, and manager interface abbreviations, respectively. In Figure 1-4, the layer abbreviation **Nh** stands for LTE RRC.
- **XX** is the two-letter prefix for the Protocol Layer. The naming conventions for the primitives invoked into the layer are:
  - **<XxUiyyy>** at the upper interface
  - **<XxMiLxx>** at the management interface
- The conventions for primitives invoked from Protocol Layer are as follows:
  - **XxUiyyy** at the upper interface, where **Xx** represents the two-letter prefix of the application layer
  - **SmMiLxx** at the manager interface, where **Sm** represents the two-letter prefix of a management entity

- **Req**, **Rsp**, **Ind**, or **Cfm** represents the primitive **type**. The naming convention for interface primitive follows this order:
  - **Xx** is the layer abbreviation.
  - **Ui**, **Li**, or **Mi** is the upper, lower, or layer manager interface.
  - **YYY** is the interface abbreviation.
  - **Lxx** is the layer manager interface name for any given layer.

  In Figure 1-4, the primitive type, **ConReq,** represents Connection Request.

A management primitive invoked in the layer manager towards the Protocol Layer uses the naming convention **SmMiLxx<prim_name><prim_type>,** where **prim_name** is the appropriate primitive and **prim_type** is the appropriate primitive type. For example, to configure the Protocol Layer, the layer manager invokes **SmMiLxxCfgReq**. This primitive, when terminated in the Protocol Layer, uses the naming convention **XxMiLxx<prim_name><prim_type>,** where **prim_name** is the appropriate primitive and **prim_type** is the appropriate primitive type. The configuration primitive, **SmMiLxxCfgReq**, is terminated in the Protocol Layer as **XxMiLxxCfgReq.**

The Protocol Layer invoking a primitive towards the layer manager uses the naming convention **XxMiLxx<prim_name><prim_type>**, where **prim_name** is the appropriate primitive and **prim_type** is the appropriate primitive type. For example, the Protocol Layer invokes the primitive **XxMiLxxCfgCfm** towards the layer manager to confirm the configuration request. This primitive, when terminated in the layer manager, uses the naming convention **SmMiLxx<prim_name><prim_type>**, where **prim_name** is the appropriate primitive and **prim_type** is the appropriate primitive type. The configuration confirm primitive **XxMiLxxCfgCfm** is terminated in the layer manager as **SmMiLxxCfgCfm**.

1.5 Data Types

Table 1-2 lists the primitive data types and their sizes.

Table 1-2: Primitive data types and their sizes

Mnemonic	# of 8 bit bytes	Sign
S8	**1**	**Signed**
U8	**1**	**Unsigned**
S16	**2**	**Signed**
U16	**2**	**Unsigned**
S32	**4**	**Signed**
U32	**4**	**Unsigned**
PTR	**as required**	**Unsigned**

The size of **PTR** depends on the specific machine in which the software is ported.

1.6 Common Identifiers

Common identifiers (ID) are used across the interfaces provided by the Protocol Layer. Also, some parameters are common across specific interfaces. This section describes some of the common identifiers.

1.6.1 Entity ID

In TAPA, each protocol layer is assigned a unique ID, known as the entity ID. System services uses this ID to determine the specific layer from which a primitive was invoked and to which layer the primitive is destined.

1.6.2 Instance ID

In TAPA, the instance ID distinguishes between multiple instances of a single protocol layer. Currently, TAPA layers are designed to have global variables. Multiple instances can be used only when each instance (in a given logical processor, only one instance of a particular layer is permitted) has its own address space, that is, in host-based environments like SUN Solaris or Windows NT. In embedded systems such as VxWorks or pSOS, where the address space is usually shared across multiple tasks, multiple TAPA layer instances cannot be used.

1.6.3 Processor ID

In TAPA, the processor ID identifies a logical processor on which the layer is executing. This can be the logical processor ID of a process in a multiprocessing environment, or it can be used to identify a specific processor on which the layer is executing.

1.7 Common Structures

Parameters have the following generic calling sequence:

XxUiyyy<prim_name><prim_type> (pst, suId/spId, suConnId/spConnId...)

The parameters described in the following subsections frequently occur.

1.7.1 Post

In TAPA, a system consists of multiple TAPA entities or tasks. The post structure:

- Is exchanged as the first parameter along with the primitive between various tasks.
- Contains the information required to identify source and destination TAPA tasks. When the interface is loosely coupled between the source and the destination layers, the source layer provides information required by the system services to route the message buffer to the correct destination layer. In the destination layer, the post structure is used only to verify the identity of the source and the specific primitive.
- Assigns each message with a priority and the priority-based message scheduler make use of it.

```
typedef struct pst          /* Parameters for SPstTsk */
{
   ProcId    dstProcId;     /* Destination processor ID (U16) */
   ProcId    srcProcId;     /* Source processor ID (U16) */
   Ent       dstEnt;        /* Destination entity (U8) */
   Inst      dstInst;       /* Destination instance (U8) */
   Ent       srcEnt;        /* Source entity (U8) */
   Inst      srcInst;       /* Source instance (U8) */
   Prior     prior;         /* Priority (U8) */
   Route     route;         /* Route (U8) */
   Event     event;         /* Event (U8) */
   Region    region;        /* Region (U8) */
   Pool      pool;          /* Pool (U8) */
   Selector  selector;      /* Selector (U8) */
   CmIntfVer intfVer;       /* Interface version */
} Pst;
```

The following table lists the structure parameters and describes their functions.

Parameter	Description
dstProcId	Destination Processor ID. Identifies the logical processor of the destination task to which the primitive must be delivered. In a service user SAP, layer manager configuration provides the value. In a service provider SAP, this value is obtained during the bind procedures. Use this parameter only when the source and destination layers are loosely coupled. Also, see Section 1.6.3.

Parameter	Description
srcProcId	Source Processor ID. Identifies the logical processor of the source task which originated the primitive. This value is obtained when system services initializes the layer. Also, see Section 1.6.3.
dstEnt	The Entity ID of the destination layer. In a service user SAP, layer manager configuration provides this value. In a service provider SAP, this value is obtained during the bind procedures. Also, see Section 1.6.1.
dstInst	The Instance ID of the destination layer. In a service user SAP, layer manager configuration provides this value. In a service provider SAP, this value is obtained during the bind procedures. Also, see Section 1.6.2.
srcEnt	The Entity ID of the originating layer. This value is obtained when system services initializes the layer. Also, see Section 1.6.1.
srcInst	The Instance ID of the originating layer. This value is obtained when system services initializes the layer. Also, see Section 1.6.2.
prior	Message priority. This value is provided at the SAP configuration. This value can be used to implement a message priority mechanism. Use of this field is implementation-dependent. See Section 2.4.1 of *System Services Interface Service Definition* for more details.
route	Message route. This field can be used in addition to the IDs above to route a message from the originating layer to the destination layer. This value is provided at the SAP configuration. See Section 2.4.2 of *System Services Interface Service Definition* for more details.
event	Event type. This parameter identifies the type of primitive generated. The IDs for all the primitives generated at an interface are defined in the corresponding interface header files.
region	Region ID. This is the ID of the dynamic memory region from which the primitive message buffers are allocated. The parameter is provided during configuration.
pool	Pool ID. This parameter identifies the dynamic memory pool from which the primitive message buffers are allocated. This parameter is provided during configuration.
selector	Coupling ID. This parameter identifies whether the interface between the source and destination layers is loosely or tightly coupled. When there are multiple destination layers, this parameter also selects the specific destination layer for the tightly coupled operation. This parameter is provided during configuration.
intfVer	Interface version is used in the protocol layer supporting the Rolling Upgrade feature to determine the version.

1.7.2 SuId and SpId

When the service user calls a primitive, **SpId** identifies the SAP number in the service provider to which the primitive is directed.

When the service provider calls a primitive, **SuId** identifies the SAP number in the service user to which the primitive is directed.

SpId and **SuId** associate a received primitive with a specific SAP in a layer. The service user and service provider define separate IDs, because each ID may have a different set of SAP control blocks for which different IDs may be supplied.

1.7.3 SuConnId and SpConnId

These fields are utilized only in connection-oriented protocols.

When the service user calls a primitive, **SpConnId** is the local ID for a connection in the service provider SAP to which the primitive is directed.

When the service provider calls a primitive, **SuConnId** is the local ID for a connection in the service user SAP to which the primitive is directed.

Both **SpConnId** and **SuConnId** IDs associate a received primitive in a SAP. Separate IDs are defined in the service user and service provider, for each to use a locally unique ID.

1.7.4 Date and Time

The date and time structure is filled by Protocol Layer when sending back the statistics values. It has the following format:

```
typedef struct dateTime
{
   U8  month;
   U8  day;
   U8  year;
   U8  hour;
   U8  min;
   U8  sec;
   U8  tenths;
#ifdef SS_DATETIME_USEC
   U32 usec;
#endif
} DateTime;
```

The following table lists the structure parameters and describes their functions.

Name	Description and Allowable Values
month	Month. The allowable values are: 1 to 12.
day	Day. The allowable values are: 1 to 31.
year	Year. The allowable values are: 1 to 255. Trillium products represent year as an offset from the year 1900. For example, 1999 is represented as 99, 2003 is represented as 103, and so on. Trillium products use one byte to store the year value. Thus, the years 1900 to 2155 can be represented without causing an overflow. Customers can add the value 1900 to the value provided by the Trillium product to get the 4 digit year. Refer to note below.
hour	Hour. The allowable values are: 0 to 23.
min	Minute. The allowable values are: 0 to 59.
sec	Second. The allowable values are: 0 to 59.
tenths	Tenths of a second. The allowable values are: 0 to 9.
usec	Micro seconds. The allowable values are: 0 to 999999.

Note: The **DateTime** structure is defined in the file **gen.x**. The layer management can compute the absolute year field by adding 1900 to the value returned by the Trillium product. The following C code shows how this can be done:

```
U32         absYear;              /* Absolute year */
DateTime    tTime;
(Void)SGetDateTime(&tTime);
absYear = tTime.year + 1900;
```

1.7.5 Header

The header structure carries all common information required to identify the management primitive.

Each management primitive has a single parameter consisting of a header structure, which is followed by a structure and is specific to the type of primitive invoked.

```
typedef struct header      /* Header */
{
   U16      msgLen;        /* Message length */
   U8       msgType;       /* Message type */
   U8       version;       /* Version */
   U16      seqNmb;        /* Sequence number */
   EntityId entId;         /* Entity ID */
   ElmntId  elmId;         /* Element ID */
#ifdef LMINT3
   TranId   transId;       /* Sequence number */
   Resp     response;      /* Structure containing response */
#endif /* LMINT3 */
} Header;
```

The following table lists the fields and defines their functions.

Field	Description and Allowable Values
msgLen **msgType** **version** **seqNmb** **entId**	Unused.
elmId	Identifies the layer-specific resource to which the management primitive applies. Also, see Section 1.7.5.1.
transId	Used by the layer manager to correlate the request primitives generated to the Protocol Layer and the confirm responses generated by the Protocol Layer to the layer manager. Correlation is required only if the layer manager generates multiple management requests with the same **msgType** value. Otherwise, this field may be ignored. Allowable values: 1 to (2^{32}-1).
response	Filled by the layer manager when invoking primitives towards the Protocol Layer. This information is used by the Protocol Layer in the response primitive sent to the layer manager. Also, see Section 1.7.5.2.

1.7.5.1 ElmntId

This field identifies the specific resource to which the management primitive pertains. Only the **Elmnt** field of this structure is mandatory. The remaining fields: **ElmntInst1**, **ElmntInst2**, and **ElmntInst3** are not used by the Protocol Layer.

```
typedef struct elmntId
{
   Elmnt       elmnt;
   ElmntInst1  elmntInst1;
   ElmntInst2  elmntInst2;
   ElmntInst3  elmntInst3;
} ElmntId;
```

The following table lists the fields and defines their functions.

Field	Description and Allowable Values
elmnt	Identifies the layer resource. The use of each resource is explained in the description of the individual management primitives. Allowable Values: **STGEN**: Used in management primitives to refer to the entire layer. **STTSAP**: Used to manage the Transport SAP (TSAP). **STSID**: Used to retrieve the software identification information. **STGRTSAP**: Used in control request to refer to a group of TSAPs. **STSOCKINIT**: Used for socket initialization. **Note**: Protocol Layer can define other values specific to a protocol in addition to the values mentioned here.
elmntInst1	Used in some of the layers. If **LMINT3** flag is enabled, it is not being used.
elmntInst2 **elmntInst3**	Unused.

1.7.5.2 Resp

The layer manager uses this field to convey information to the Protocol Layer of the parameters which must be filled in the confirmation primitive by the layer. The data structure and its parameters are:

```
typedef struct resp
{
   Selector  selector;      /* Selector */
   Priority  priority;      /* Priority */
   Route     route;         /* Route */
   Memory    mem;           /* Memory/pool */
}Resp;
```

The following table lists the fields and defines their functions.

Field	Description and Allowable Values
selector	Allowable values are: **LXX_SMLC**: indicates the layer manager is loosely coupled. **LXX_SMTC**: indicates the layer manager is tightly coupled. Also, see Section 1.7.1.
priority	Allowable values are PRIOR0, PRIOR1, PRIOR2 and PRIOR3. Also, see Section 2.4.1 of *System Services Interface Service Definition*.
route	Allowable value are: **RTESPEC** for the conventional Trillium layer **RTEPROTO** for the fault tolerant Trillium layer. Also, see Section 2.4.2 of *System Services Interface Service Definition*.
mem	Identifies the region and pool values from which the message buffer to be sent to the layer manager must be allocated by the Protocol Layer. Also, see Section 1.7.6.

1.7.6 Memory

This parameter specifies the memory region and pool values.

```
typedef struct mem         /* Memory */
{
   Region region;          /* Region */
   Pool pool;              /* Pool */
   U16 spare;              /* Spare for alignment */
}mem;
```

Note: The description and allowable values for the region and pool fields are the same as in the description of the **Post** structure in Section 1.7.1. The **spare** field is used for alignment only.

1.8 Common Interface Structures

1.8.1 Tokens

The tokens represent the values of individual fields in the IEs. The token types are defined, based on the arithmetic value type they need to represent. Tokens have a PRESENT flag to indicate whether the value in the **`val`** field is valid. This must represent the optional fields in the IEs.

Also, to prevent the addition of compiler-generated padding to the event structures tokens contain **`spare`** fields, so that the token is aligned on a 32-bit or 64-bit boundary (the **`ALIGN_64BIT`** compile-time option enables a 64-bit alignment). Compiler-generated padding interferes with the decoding/encoding operations for the SDUs and must be strictly avoided.

1.8.1.1 Token U8

This token represents the IE fields—up to 8 bits.

```
typedef struct tknU8          /* Token U8 */
{
    U8  pres;                  /* Present flag */
    U8  val;                   /* Value */
    U16 spare1;                /* For alignment */
    #ifdef ALIGN_64BIT
        U32 spare2;            /* For 64-bit alignment */
    #endif
} TknU8;
```

1.8.1.2 Token U16

This token represents the 16-bit IE fields.

```
typedef struct tknU16         /* Token U16 */
{
    U8  pres;                  /* Present flag */
    U8  spare1;                /* For alignment */
    U16 val;                   /* Value */
    #ifdef ALIGN_64BIT
         U32 spare2;           /* For 64-bit alignment */
    #endif
} TknU16;
```

1.8.1.3 Token U32

This token represents the 24- or 32-bit IE fields.

```
typedef struct tknU32         /* Token U32 */
{
    U8  pres;                  /* Present flag */
    U8  spare1;                /* For alignment */
    U16 spare2;                /* For alignment */
    U32 val;                   /* Value */
} TknU32;
```

1.8.1.4 Token String 4

This token represents the IE fields that are smaller than four octets. Typically, these fields represent the string values, as opposed to the arithmetic values, such as the address signals in the called party number. It is defined as:

```
typedef struct tknStr4       /* Token string */
{
    U8  pres;                 /* Present flag */
    U8  len;                  /* Length */
    U16 spare1;               /* For alignment */
    #ifdef ALIGN_64BIT
        U32 spare2;           /* For 64-bit alignment */
        U8 val[8];            /* String value */
    #else
        U8 val[4];            /* String value */
    #endif
} TknStr4;
```

1.8.1.5 Token Presence

This token presence represents the presence of the IE. Only the **pres** field is used in the **TknPres**; the **val** field is not used to represent anything.

```
typedef TknU8 TknPres;
```

1.8.1.6 Token Octet String XL

This token string represents the variable-length octet strings. The length field specifies the length of the value part, in octets.

```
typedef struct tknStrOSXL /* Token string extra long */
{
    U8  pres;              /* Present flag */
    U8  spare1;            /* For alignment */
    U16 len;               /* Length */
    #ifdef ALIGN_64BIT
        U32 spare2;       /* For 64-bit alignment */
    #endif
    U8 *val;               /* String value (use allocated memory) */
}TknStrOSXL
```

1.8.1.7 Token Bit String 32

This token represents the IE fields that are less than four octets (32 bits). Typically, these fields represent the string of bits. It is defined as:

```
typedef TknStr4 TknBStr32;
```

1.8.1.8 Token Bit String XL

This token string represents the variable-length bit strings. The length field specifies the length of the value part in bits.

```
typedef TknStrOSXL TknStrBSXL;
```

1.8.1.9 Token Bool

This token represents a Boolean type.

```
typedef TknU8   TknBool;
```

1.8.1.10 Token Null

This token represents a Null type.

```
typedef TknU8   TknNull;
```

1.8.1.11 Token Enum

This token represents an enumerated type.

1.9 Return Value

All interface primitives return a signed 16-bit value. This return code usually takes two values:

```
#define         ROK         0       /* Success */
#define         RFAILED     1       /* Failed */
```

Sometimes, other non-zero codes are returned to indicate specific failure conditions (such as, out of resources).

When the primitive supplier and the primitive caller are tightly coupled, the return code for the called primitive is supplied directly by the layer in which the primitive terminates. A meaningful value can be returned to the primitive caller.

However, when the primitive supplier and primitive user are loosely coupled, the called primitive terminates when it is packed and posted in a message to the primitive supplier. At this point, the primitive has not been processed by the supplier, and any status code that needs to be returned by the supplier cannot be relayed to the caller when the called primitive returns. In this case, the return code is supplied by the interface pack and post functions.

The return code from a called primitive is not considered significant because it may have come from the caller's interface functions or from the supplier. In fact, no status is exchanged through primitive return codes. Instead, when the supplier needs to return a status to the caller, it uses an explicit primitive.

For example, if the service user calls a connect request primitive and the service provider is unable to provide the service, the service provider returns a release indication primitive to the service user, rather than just returning an **`RFAILED`** status code for the connect request primitive. Similarly, if the service provider is able to provide the service, it returns a connect confirm primitive, rather than returning an **`ROK`** status code.

In subsequent descriptions, no return codes are specified for the primitives, because they are not significant.

1.10 Handling of Error Conditions

When a primitive is received by the service user or service provider, there can be errors in processing the primitive (such as incorrect parameter values and resources unavailable). Since error conditions are not indicated by return values from primitives, they are generally handled as described below. If the error handling procedures for a primitive are different from those described below, they are mentioned explicitly in that primitive's description.

If the service provider is unable to service a primitive because of errors and is unable to locate the service user, it discards the primitive. This generates an alarm to the layer manager and no primitive is returned to the service user.

If the service provider is unable to service a primitive because of errors and is able to locate the service user, it generates an appropriate return primitive that indicates failure of the primitive previously issued. It does not generate an alarm to the layer manager in this case. For example, if the service provider is unable to service a **`XxYyXXXConReq,`** it generates an **`XxYyXXXDiscInd`** and indicates local failure.

If the service user is unable to service a primitive and is also unable to locate the service provider, it discards the primitive. The service user generates an alarm to the layer manager. In this case, no primitive is returned to the service provider.

If the service user is unable to service a primitive (for example, because of unavailable resources or a misdirected request) and it is able to locate the service provider, it may reject the primitive (for example, by returning **`XxYyXXXDiscReq`** to the service provider in response to an **`XxYyXXXConInd`**) to reject the particular connection specified in the primitive.

2

Writing Applications for Trillium's Portable Layer

2.1 Guidelines

This section highlights the aspects that must be considered when writing application, layer manager, or lower layer interfaces with Trillium's portable layer conforming to TAPA. Customers using multiple layers in a stack must consider these points while writing the stack manager and the layers interacting with the stack of Trillium layer(s).

2.1.1 Reentrancy

Trillium portable layers are not reentrant (unless specified otherwise). There must not be more than one thread executing the layer code simultaneously.

2.1.2 Coupling Options

TAPA provides the following coupling options. TAPA mandates the same coupling be used for all primitives in the same direction on an interface.

Coupling	Description
Loose	Uses an asynchronous interface, where a primitive is packed into a message and posted to the destination layer. Control returns to the source layer immediately. Later, the operating system schedules the destination layer with the posted message. The destination layer retrieves the primitive from the message and processes it. Packing and Unpacking involves some CPU cycles thereby the performance is slightly lower compared to Tight coupling. It enables the communicating layers to work as separate threads of execution.
Tight	Uses a synchronous interface, where a primitive is mapped to a direct function call into the destination layer. Control is transferred to the destination layer immediately, in a nested fashion. This function nesting results in increased stack size. The performance is slightly higher than Loose coupling as there is no need for packing and unpacking.
Light-weight Loose	Uses an asynchronous interface, where a pointer to the event structure is packed into a message and posted to the destination layer. This is applicable if and only if the sender and the receiver are in the same address space. Control returns to the source layer immediately, like Loose Coupling. Later, the operating system schedules the destination layer with the posted message. The destination layer retrieves the event structure by unpacking the pointer. It inherits the advantages of both Loose and Tight couplings as only the packing and unpacking of pointers is required. It is mainly useful at interfaces, which carry lot of information. Due to this only few layers implement this feature.

2.1.3 Buffer Management across the Interface(s)

Trillium portable layers assume that the receiver of the primitive de-allocates the allocated message buffers. This is applicable in both tight and loose coupling scenarios. If a message is received by a portable layer and it passes the message buffer (type Buffer *) to its application, then it is the application's responsibility to de-allocate the buffer when included in the primitive. Similarly, if an application sends some data in message buffer format (as defined by the primitive) then the application must not de-allocate the buffer after invoking the primitive.

Continuous Computing advises referring to the sample code for the layer regarding this scenario. A message buffer may not be presented as a parameter in the primitive, it can be presented as a field in a parameter; in this case also the rule mentioned in this section applies.

2.2 Event Structures at Interfaces

Interface message structure provides all the information required for processing the primitive in the destination layer. Some interface message structures might carry more information (parameters) depending on the Protocol Layer requirements. These interface message structures are referred to as event structures in TAPA.

2.3 Event Structure Memory Management

The size of the parameters and number of the parameters in an event structure can have very large memory requirements for some interfaces. Therefore, a separate memory management scheme is used to ensure efficient utilization of the memory for the event structures. Therefore, many members of the event structures are defined as pointers and these members must be allocated when they are initialized.

The event structure memory management scheme allocates the fixed-size blocks of memory, and any pointer parameters in the event structures are allocated in these memory blocks. The size of the block is based on the size of the biggest structure in the event. These memory blocks are referred to as event blocks in the rest of the document.

To manage the multiple event blocks from which the event parameters are allocated, each of the top-level event data structures contain **`CmMemListCp`** as the first field. The size of the event block can be configured and must be at least equal to the largest structure size at that interface. The data structures and interface functions used to manage the event blocks are described in the following sub-sections.

When these allocated event structures are passed in a primitive across the interface, the layer generating the primitive gives up ownership of the event block(s). Thus, once the primitive is generated, the layer generating the primitive must not free the event block. The freeing of the event block occurs:

1. In the primitive packing functions, if the interface between the layers is loosely coupled.
2. In the receiving layer, if the interface between the layers is tightly or light-weight loosely coupled.

The use of the event blocks and library routines at that interface are described as follows.

2.3.1 Memory List Control Point

Each event data structure contains the memory list control point structure to help manage the event blocks. The structure is:

```
typedef struct cmMemListCp
{
   CmMemList     *first; /* First entry in list  */
   CmMemList     *last;  /* Last entry in list   */
   U32           count;  /* Number of entries    */
   CmMemCb       memCb;  /* Memory Control Block */
} CmMemListCp;
typedef struct cmMemList CmMemList;
```

This table lists the fields and their allowed values:

Field	Description and allowed values	Refer to
First **Last**	Pointers to the first and last memory blocks in the linked list of allocated memory blocks. Each **CmMemList** has the previous and next pointer information to manage the list.	Section 2.3.1.1, "Memory List"
count	The actual number of memory control blocks currently allocated.	--
memCb	All the allocated information about the memory blocks.	Section 2.3.1.2, "Memory Control Block"

As shown in the data structure in Figure 2-1, **CmMemListCp** has information on the size of each event block, the start of the first event block, a pointer into the event block at which the next allocation takes place, and a double-linked list of all the allocated event blocks. The initial portion of each allocated event block contains a **CmMemList** parameter to allow the chaining of all the memory blocks that belong to an event structure. Refer to Figure 2-1 and Figure 2-2 for pictorial description.

Figure 2-1 shows the contents of the **`CmMemListCp`**.

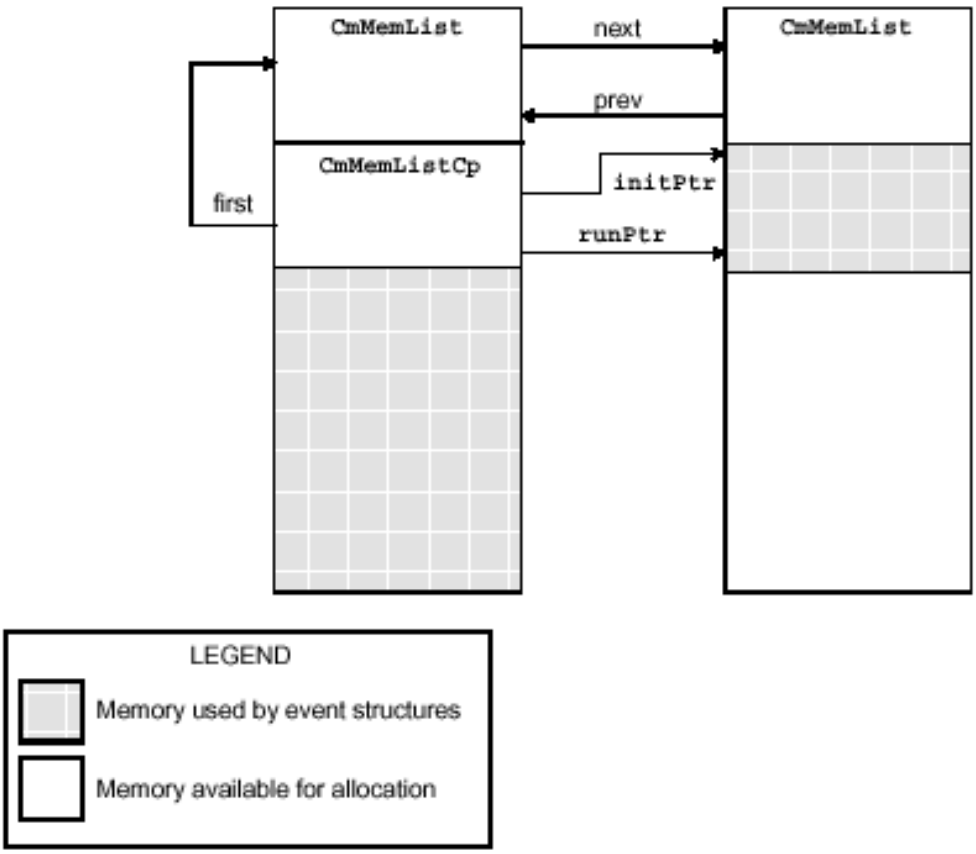

Figure 2-1: Contents of CmMemListCp

Figure 2-2 shows the contents of the **CmMemListCp** and **CmMemCb** data structure instances:

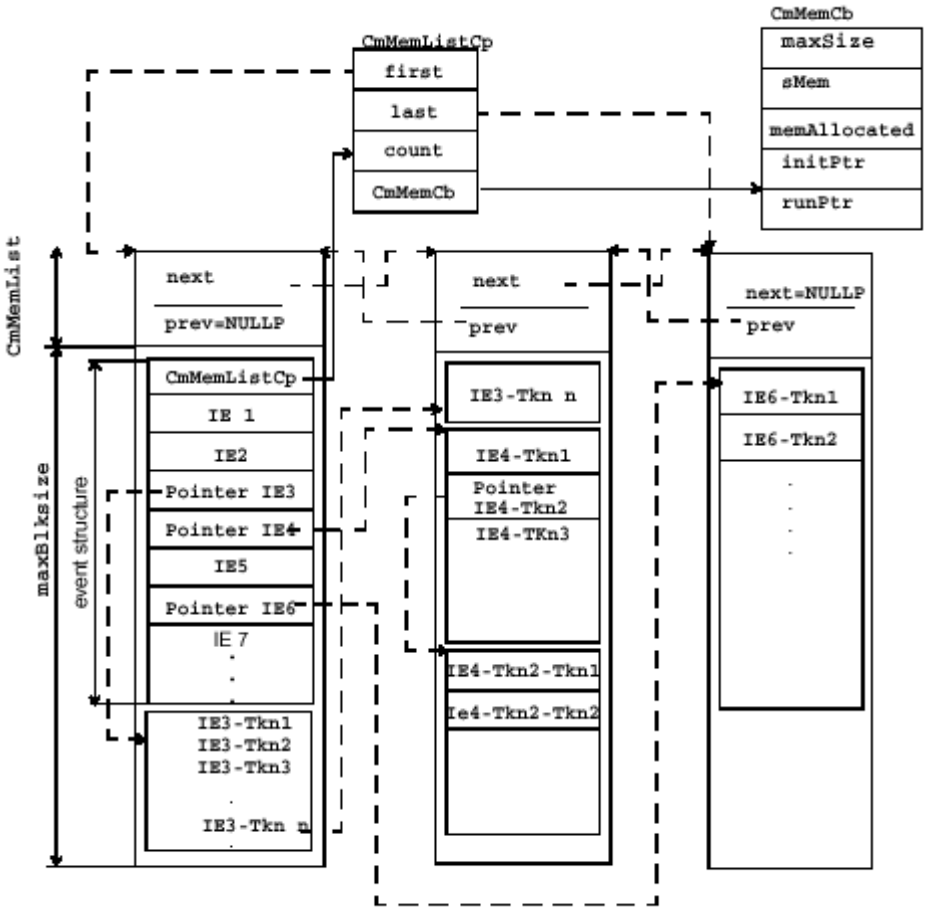

Figure 2-2: Event block of an example event structure

Figure 2-2 also shows:

- The allocation of an example event structure allocated in the first event block. It contains the information elements IE1, IE2, IE5, IE7, and the pointer (allocated) information elements IE3, IE4, and IE6.
- That information element IE3 is allocated in the same event block as that of the event structure, and in turn contains token elements **Tkn1**, **Tkn2, Tkn3**, and an allocated token element **Tknn**.
- That **IE3-Tknn** is allocated in the second event block.
- The token information of IE4.
- The links between the memory blocks and the first and last members of **CmMemListCp**.

2.3.1.1 Memory List

The structure of the memory list is:

```
typedef struct cmMemList
{
   CmMemList *next;   /* Next */
   CmMemList *prev;   /* Previous */
   Size       size;   /* Block size */
} CmMemList;
```

2.3.1.2 Memory Control Block

The structure of the memory control block is:

```
typedef struct cmMemCb
{
   Size  maxSize;      /* Size of memory chunk */
   Mem   sMem;         /* Static memory region and pool */
   U32   memAllocated; /* Amount of memory already allocated */
   PTR   initPtr;      /* Initial pointer */
   PTR   runPtr;       /* Start of avaliable memory chunk */
} CmMemCb;
```

This table lists the fields and their allowed values:

Field	Description and allowed values
maxSize	Specifies the size of the individual event blocks allocated. Allowed values: 0 to (2^{32} - 1).
sMem	Identifies the memory region and memory pool from which the event blocks are allocated.
memAllocated	Maintains the total amount of memory allocated in the current event block. The allowed values: 0 to **maxSize**.
initPtr	Indicates the start of the current event block from which the memory is currently allocated.
runPtr	Indicates the pointer into the current event block at which the next allocation occurs.

2.4 Event Structure Allocation

Name

cmAllocEvnt

Synopsis

```
PUBLIC S16 cmAllocEvnt (evntSize, maxBlkSize, sMem, ptr)
Size     evntSize;
Size     maxBlkSize;
Mem      *sMem;
Ptr      *ptr;
```

Parameters

Field	Description and allowed values
evntSize	Size of the event structure to be allocated. This can be obtained by using the **sizeof** operation in the event structure.
maxBlkSize	Size of the event block to be allocated from which the other parameters in the event structure are allocated.
sMem	Memory region and memory pool from which the library must allocate the memory.
ptr	Pointer to the event structure to be allocated. Upon returning from this function, the library allocates the required event structure, and the **CmMemListCp** parameter in the event structure is initialized for further operation.

Description

This function must be the first call for each event (the event structure has the **CmMemListCp** as its first member) structure initialization in the primitive. The memory management library allocates an event block of size **maxBlkSize** and imposes the requested event structure on this event block, with the **CmMemListCp** field in the event structure initialized to the correct values.

The library operates only on the **CmMemListCp** portion of the event structure. The pointer members are allocated in the same event block, until the requested size cannot be fulfilled by the event block. A new event block is then allocated.

The **initPtr**, **runPtr**, and **memAllocated** fields of the **CmMemCb** member of the **CmMemListCp** structure are updated. The first and last parameters of the **CmMemListCp** structure are also updated.

Return Values

Value	Description
ROK	Returned when memory allocation is successful and errors are not encountered.
RFAILED	Returned when either the **evntSize** parameter is greater than the **maxBlkSize** parameter, or when the event block allocation fails.

2.5 Allocate Parameters Memory

Name

cmGetMem

Synopsis

```
PUBLIC S16 cmGetMem (memPtr, size, allocPtr)
Ptr    memPtr;
Size   size;
Ptr    *allocPtr;
```

Parameters

Field	Description and allowed values
memPtr	The pointer parameter returned by the **cmAllocEvent** function in the **ptr** parameters.
size	Specifies the size of the event parameters to be allocated.
allocPtr	Pointer to the event parameters to be allocated.

Description

Once the event structure is allocated using the **cmAllocEvnt** function, any other parameters in the event structure that require memory allocation are allocated using this function. The library checks to refer whether the memory requirement can be satisfied by the current event block. If so, it returns a pointer to the position in the event block at which the memory allocation can be satisfied in the **allocPtr** parameter. Otherwise, a new event block is allocated and chained to the list of event blocks on the event structure by updating the first and last parameters of the **CmMemListCp** data structure.

The pointer to the position in the newly allocated event block, at which the requested allocation can be satisfied, is returned. The **memAllocated**, **initPtr**, and **runPtr** parameters in the **CmMemCb** member of the **CmMemListCp** structure are updated.

Return Values

Field	Description
ROK	Returned when memory allocation is successful and errors are not encountered.
RFAILED	Returned when either the **size** parameter is greater than the maximum size of the event block as specified in the **cmAllocEvnt**, or when the memory allocation for a new event block fails.

2.6 Free Memory

Name

cmFreeMem

Synopsis

```
PUBLIC S16 cmFreeMem (memPtr)
Ptr    memPtr;
```

Parameters

Field	Description and allowed values
memPtr	Pointer parameter returned by the **cmAllocEvent** function in the **ptr** parameters.

Description

This function must be invoked only once to free the entire event structure. Only the event structure pointer is required as a parameter. The function traverses the list of event blocks in the event structure and frees each of them, and finally frees the event block that contains the event structure.

Return Values

Field	Description
ROK	Returned when the freeing of the memory is successful and errors are not encountered.
RFAILED	Returned when the freeing of the memory for the event block(s) fails.

2.7 Memory Library Usage

The memory library, **cmAllocEvnt**, must be used to allocate any event structures. It is the first function to be called. Any other memory to be allocated for the members within the event structures must be allocated using the **cmGetMem** function.

Note: The following convention is used to allocate and free the memory. The layer originating the message allocates the memory. The destination layer which receives the message, frees the memory. This is true for all the coupling options at the interfaces.

Refer to the following code snippet on how to use the memory library.

```
typedef struct _xxMsgIE
{
  TknPres   pres;
  TknU32    value;
} XxMsgIE;

typedef struct _xxMsgIECont
{
  TknU16    noComp;
  XxMsgIE   **msgIE;
} XxMsgIECont;

typedef struct _xxDatReqMsg1
{
  TknPres          pres;
  XxMsgIECont      msgIECont;
} XxDatReqMsg1;

typedef struct _xxReqSdus
{
   CmMemListCp     memCp;
   XxHdr           hdr;            /*!< SDU Header */
   union
   {
     XxDatReqMsg1  xxMsg1;         /*!< Message to be sent */

     /* Other messages are defined here */
   } sdu;
} XxReqSdus;
```

Example 1: Originator is sending the event structure to the receiver in an interface primitive.

```
 S16 tstSendMsg()
{
    XxReqSdus  *reqSdus;
    Size       maxBlkSize;
    Mem        sMem;
    XxMsgIE    **msgIePtr;
    XxMsgIE    *msgIe;

    reqSdus = NULLP;
    msgIe = NULLP;
    msgIePtr = NULLP;
    maxBlkSize = XX_MEM_PDU_SIZE;
    sMem.region = DFLT_REGION;
    sMem.pool = DFLT_POOL;
    /* allocate the memory for the event struct */
    if(cmAllocEvnt(sizeof(XxReqSdus), maxBlkSize, &sMem,
                                        (Ptr*)&reqSdus) != ROK)
   {
      RETVALUE(RFAILED);
   }
   /* Fill the header portion */
   xxTstFillSduHdr(&(reqSdus->hdr));

   /* Allocate memory for the Ptr to MsgIEs */
  if((cmGetMem(reqSdus, 1 * sizeof(PTR), (Ptr*)&msgIePtr)) != ROK)
  {
    cmFreeMem(&(reqSdus->memCp));
  }

   /* Allocate the memory for the msgIe */
  if((cmGetMem(reqSdus, sizeof(XxMsgIE), (Ptr*)&msgIe)) != ROK)
  {
    cmFreeMem(&(reqSdus->memCp));
  }

  /* Fill the msgIe */
  XxTstFillMsgIE(msgIe);

  msgIePtr[0] = msgIe;

  reqSdus->sdu.xxMsg1.msgIECont.msgIE = msgIePtr;
  reqSdus->sdu.xxMsg1.msgIECont.noComp.pres = PRSNT_NODEF;
  reqSdus->sdu.xxMsg1.msgIECont.noComp.val = 1;

  /* Send the message with the event structure to destination here */

}
```

Example 2: After processing the primitive, the receiver is freeing the event structure.

```
S16  tstPrcMsg(reqSdu)
{
   /* Process the message with event structure */

   /* Free the memory associated with the event structure */
  cmFreeMem(&(reqSdus->memCp));
}
```

2.8 Stack Initialization Guidelines

This section describes the steps to initialize and configure the Protocol Layer(s) in a Trillium stack.

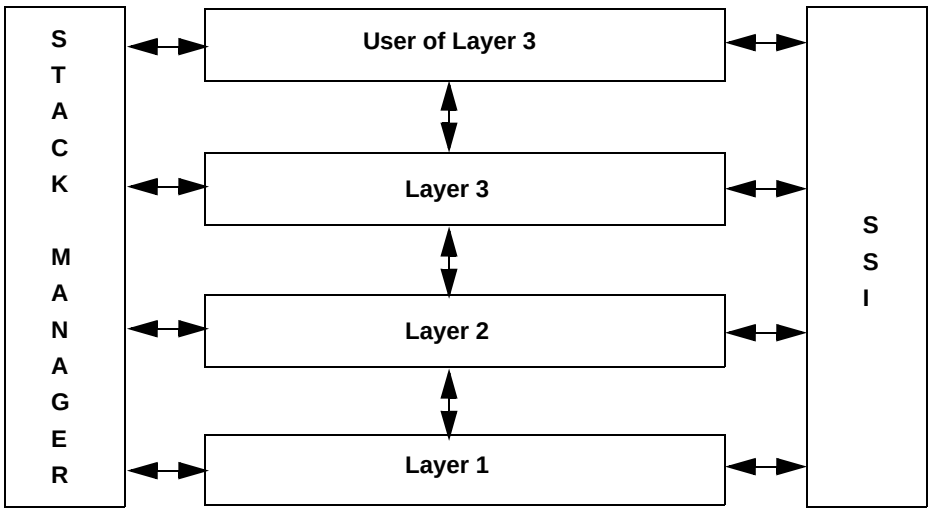

Figure 2-3: Sample stack with three Trillium layers

Consider three Trillium layers to be initialized and configured as shown in Figure 2-3. The customer develops the application, which is the User of Layer 3 and also the Stack Manager to configure and control the stack.

The following steps must be performed:

1. The Stack Manager registers with SSI all the layers in the stack including the application.
2. From the Stack Manager, configure Layer 1, Layer 2, and Layer 3 as defined in the respective Service Definition document.

3. From the Stack Manager:

   a) Send Bind Control Request to Layer 2; to bind the lower SAPs with the upper SAPs of Layer 1.

   2) Send Bind Control Request to Layer 3; to bind the lower SAPs with the upper SAPs of Layer 2.

   3) Send Bind Control Request to User of Layer 3; to bind the lower SAPs with the upper SAPs of Layer 3.

Now, the stack is fully configured and ready to process.

2.9 Stack Shutdown Guidelines

This section describes the steps to shutdown the Protocol Layer(s) in a Trillium stack.

Refer to Figure 2-3 and perform the following steps:

1. The User of Layer 3 must send the unbind request to all the associated SAPs of Layer 3 as defined in the respective Interface Service Definition document.
2. From the Stack Manager:

   a) Send Unbind Control Request to Layer 3; to unbind its lower SAPs from the upper SAPs of Layer 2.

   2) Send Unbind Control Request to Layer 2; to unbind its lower SAPs from the upper SAPs of Layer 1.
3. From the Stack Manager; send Shutdown Control Request to Layer 3, Layer 2, and Layer 1 respectively.

Now, the stack is deactivated and non-operational.

References

Refer to the following documents for more information:

System Services Interface Service Definition, version 3.12a, Continuous Computing Corporation (p/n 1111001).