Dialogic

Dialogic® Converged Services Platform - SwitchKit® Development Environment

Programmer's Guide

February 2008

Copyright and Legal Disclaimer

All contents of this document are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries. Reasonable effort is made to ensure the accuracy of the information contained in the document. However, due to ongoing product improvements and revisions, Dialogic Corporation and its subsidiaries do not warrant the accuracy of this information and cannot accept responsibility for errors or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS EXPLICITLY SET FORTH BELOW OR AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic Corporation or its subsidiaries may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic Corporation or its subsidiaries do not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic Corporation or its subsidiaries. More detailed information about such intellectual property is available from Dialogic Corporation's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. The software referred to in this document is provided under a Software License Agreement. Refer to the Software License Agreement for complete details governing the use of the software.

Dialogic Corporation encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Dialogic, Dialogic Pro, Brooktrout, Cantata, SnowShore, Eicon, Eicon Networks, Eiconcard, Diva, SIPcontrol, Diva ISDN, TruFax, Realblocs, Realcomm 100, NetAccess, Instant ISDN, TRXStream, Exnet, Exnet Connect, EXS, ExchangePlus VSE, Switchkit, N20, Powering The Service-Ready

Dialogic Product Line Warranty

Unless otherwise stated in an applicable product purchase agreement between the Customer and Dialogic, Dialogic warrants that during the Warranty Period, products will operate in substantial conformance with Dialogic's standard published documentation accompanying the product. If a product does not operate in accordance therewith during the Warranty Period, the Customer must promptly notify Dialogic. Dialogic, at its option, will either repair or replace the product without charge. The Customer has the right, as their exclusive remedy, to return the product for a refund of purchase price or license fee if Dialogic is unable to repair or replace it.

Warranty Period

In the event that you have no signed agreement setting out a warranty period, the Warranty Period shall be the standard warranty period set out on www.dialogic.com on the date of your purchase of the product.

The Warranty Period begins on the date of shipment of any products or software by Dialogic.

The Warranty Period for repaired, replaced or corrected products and software shall be coterminous to the Warranty Provided for the original products or software purchased.

To report warranty claims, Customer may contact Dialogic via email at techsupport@cantata.com or call (781) 433-9600.

Warranty Provisions

A. During the Warranty Period, Dialogic warrants to Customer only that:

(i) Products manufactured by Dialogic (including those manufactured for Dialogic by an original equipment manufacturer) will be free from defects in material and workmanship and will substantially conform to specifications for such products;

(ii) software developed by Dialogic will be free from defects which materially affect performance in accordance with the specifications for such software. With respect to products or software or partial assembly of products furnished by Dialogic but not manufactured by Dialogic, Dialogic hereby assigns to Customer, to the extent permitted, the warranties given to Dialogic by its vendors of such items.

B. If, under normal and proper use, a defect or non conformity appears in warranted products or software during the applicable Warranty Period and Customer promptly notifies Dialogic in writing during the applicable warranty period of such defect or non conformance, and follows Dialogic's instructions regarding return of such defective or non conforming Product or Software, then Dialogic will, at no charge to Customer, either:

(i) repair, replace or correct the same at its manufacturing or repair facility or

(ii) if Dialogic determines that it is unable or impractical to repair, replace or correct the product or software, provide a refund or credit not to exceed the original purchase price or license fee.

C. No product or software will be accepted for repair or replacement without the written authorization of and in accordance with instructions from Dialogic. Removal and reinstallation expenses as well as transportation expenses associated with returning such product or software to Dialogic shall be borne by Customer. Dialogic shall pay the costs of transportation of the repaired or replaced product or software to the destination designated in the original Order. If Dialogic determines that any returned product or software is not defective, Customer shall pay Dialogic's costs of handling, inspecting, testing and transportation. In repairing or replacing any product, part of product, or software medium under this warranty, Dialogic may use new, remanufactured, reconditioned, refurbished or functionally equivalent products, parts or software media. Replaced products or parts shall become Dialogic's property.

D. Dialogic makes no warranty with respect to defective conditions or non conformities resulting from any of the following: Customer's modifications, misuse, neglect, accident or abuse; improper wiring, repairing, splicing, alteration, installation, storage or maintenance performed in a manner not in accordance with Dialogic's or its vendor's specifications, or operating instructions; failure of Customer to apply Dialogic's previously applicable modifications or corrections; or items not manufactured by Dialogic or purchased by Dialogic pursuant to its procurement specifications. Dialogic makes no warranty with respect to products which have had their serial numbers removed or altered; with respect to expendable items, including, without limitation, fuses, light bulbs, motor brushes and the like; or with respect to defects related to Customer's data base errors. Improper packaging of product for repair will not be covered under this warranty agreement. No warranty is made that software will run uninterrupted or error free.

E. Warranty does not include:

a) Dialogic's assistance in diagnostic efforts;

b) access to Dialogic's Technical Support web sites, databases or tools;

c) product integration testing;

d) on-site assistance; or

e) product documentation updates.

These services are available either during or after warranty at Dialogic's published prices.

F. THE FOREGOING WARRANTIES ARE EXCLUSIVE & ARE GRANTED IN LIEU OF ALL OTHER EXPRESS & IMPLIED WARRANTIES (WHETHER WRITTEN, ORAL, STATUTORY OR OTHERWISE), INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. CUSTOMER'S SOLE AND EXCLUSIVE REMEDY AND DIALOGIC'S SOLE OBLIGATION HEREUNDER, SHALL BE TO REPAIR, REPLACE, CREDIT OR REFUND AS SET FORTH ABOVE.

G. IN NO EVENT SHALL DIALOGIC, ITS DIRECTORS, OFFICERS, EMPLOYEES, AGENTS OR AFFILIATES, BE LIABLE FOR ANY COSTS OR DAMAGES ARISING DIRECTLY OR INDIRECTLY FROM YOUR USE OF ANY PRODUCT INCLUDING ANY INDIRECT,

INCIDENTAL, SPECIAL, EXEMPLARY, MULTIPLE, PUNITIVE OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, WHETHER BASED ON CONTRACT, TORT (INCLUDING NEGLEGENCE), STRICT LIABILITY OR OTHER LEGAL THEORY, EVEN IF DIALOGIC, OR ANY OF ITS DIRECTORS, OFFICERS, EMPLOYEES, AGENTS OR AFFILIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY EVENT, DIALOGIC'S CUMULATIVE LIABILITY TO YOU FOR ANY AND ALL CLAIMS RELATING TO THE USE OF ANY PRODUCT SHALL NOT EXCEED THE TOTAL AMOUNT OF THE PURCHASE PRICE OR LICENSE FEES PAID TO DIALOGIC FOR SUCH PRODUCT.

H. CUSTOMER AND DIALOGIC HEREBY WAIVE THEIR RIGHT TO TRIAL BY JURY TO THE FULLEST EXTENT PERMITTED BY LAW IN CONNECTION WITH ALL CLAIMS ARISING OUT OF OR RELATED TO THIS WARRANTY, THE PRODUCTS COVERED HEREBY OR THE PERFORMANCE OF ANY PARTY HEREUNDER.

I. THIS WARRANTY SHALL BE CONSTRUED UNDER AND GOVERNED BY THE LAWS OF THE COMMONWEALTH OF MASSACHUSETTS WITHOUT GIVING EFFECT TO ANY CHOICE OR CONFLICT OF LAW PROVISION OR RULE (WHETHER OF THE COMMONWEALTH OF MASSACHUSETTS OR ANY OTHER JURISDICTION) THAT WOULD CAUSE THE APPLICATION OF THE LAWS OF ANY JURISDICTION OTHER THAN THE COMMONWEALTH OF MASSACHUSETTS. CUSTOMER SPECIFICALLY AND IRREVOCABLY CONSENTS TO THE PERSONAL AND SUBJECT MATTER JURISDICTION AND VENUE OF THE FEDERAL AND STATE COURTS OF THE COMMONWEALTH OF MASSACHUSETTS AND SUCH COURTS SHALL HAVE EXCLUSIVE JURISDICTION WITH RESPECT TO ALL MATTERS CONCERNING THIS WARRANTY OR THE ENFORCEMENT OF ANY OF THE FOREGOING.

J. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM JURISDICTION TO JURISDICTION.

About this Publication

Purpose

This documentation provides guidelines for using the Dialogic® CSP.

Safety Labels

The following Safety labels may appear in this information product to alert customers to avoidable hazards. The following are in the order of priority:

DANGER

Danger indicates the presence of a hazard that will cause death or severe personal injury if the hazard is not avoided.

WARNING

Warning indicates the presence of a hazard that can cause death or severe personal injury if the hazard is not avoided.

CAUTION

Caution indicates the presence of a hazard that will or can cause minor personal injury or property damage if the hazard is not avoided. Caution can also indicate the possibility of data loss, loss of service, or that an application will fail.

Conventions used

This information product uses the text conventions explained below. In addition, hexadecimal numbers are preceded by a zero and small “x.” For example, the decimal number 15 is represented in hexadecimal as 0x0F.

Convention	Description
. . .	A horizontal ellipsis in an API message indicates fields of variable length.
:	A vertical ellipsis in an API message indicates that a block of information is repeated or is variable.
n	The letter *n* is a generic placeholder for a number.
`Sans serif mono space`	Indicates a command name, option, input, output, non-GUI error, and system messages.
`Sans serif monospace italic`	Indicates a parameter name in an input message. Example: `move *.dot a: c:` *`-s`* The *`-s`* is the parameter.
Serif italic	Indicates the name of a book, chapter, path, file, or API message. Example: *`UserDirectory/Config.exe`*
Boldface	Indicates keyboard keys, key combinations, and command buttons Example: **Ctrl+Alt+Del**
Sans serif boldface	Identifies text that is part of a graphical user interface (GUI). Example: Go to the **Configuration** menu and select **Card**->**Span Configuration**

Contents

1 SwitchKit® Application Programming Interface

Purpose

The SwitchKit® Application Programming Interface (API) provides a high-level interface between the application and the EXS® API, to facilitate rapid switch-to-application integration. This straightforward C/C++ language API is based on industry standards, and provides automatic configuration and redundancy control, descriptive logs, and error messages. It extracts the EXS API suite into a C/C++ Library format. This feature enables you to support EXS messaging as well as administrative messages (between the application and the Low-Level Communicator (LLC) with all the benefits of a high-level language API.

The power of SwitchKit allows you to concentrate your efforts on building your telephony applications, without dealing with complex configuration requirements or administrative tasks. It enhances applications by providing increased reliability and seamless integration of multiple applications and multiple hosts. The flexibility of SwitchKit allows you to have control over low-level administrative tasks. In short, SwitchKit lets you address the call processing aspects of your solutions as you see fit, giving you the power to implement new services easily, quickly, and profitably.

Reduced Development Time

SwitchKit significantly reduces development time. You do not have to be concerned with management APIs and can focus on only the messages that relate to call control.

SwitchKit not only eliminates the need to develop switch management services, but it is simple to modify and it also provides a robust, feature-rich management system with real-time monitoring and system set-up. Providing switch management separate from the switching platform and applications allows the system to remain modular. Therefore, upgrades, modifications, and customizations to any single module can be implemented without affecting the other modules, or requiring subsequent changes.

Openness and Programmability

SwitchKit API adds considerable openness and programmability to the CSP. It provides access to the EXS API through C structures or C++ classes. Programming through the SwitchKit API allows the you to focus on functional parts of the code without concern for system-level details such as checksums and message length. This translation reduces the overall amount of code necessary to program an application and simplifies commands from an internal hex format into English-like statements.

Asynchronous Call Model

SwitchKit's asynchronous call-processing model uses common state machine call flows to allow you to easily modify call programming. SwitchKit's call processing model uses stacks of event handlers, which allow events associated with the call to "trickle down" the stack until finding an applicable command. This stacking allows the code to remain modular, enabling new handlers to be added for new features and enhancements, and/or to handle unique protocols in different countries or systems.

Inter-Application Messaging

Another feature of SwitchKit API that significantly reduces programming time is Inter-Application Messaging (IAM). IAM allows applications, and components within them, to communicate using SwitchKit API messages. Without SwitchKit, you would need to write your own interfaces for communications between your applications, and from applications to other components such as a billing system. However, SwitchKit allows you to use the same messaging interface for all communication, which reduces complexity, development time, and training.

Because SwitchKit supports a wide variety of operating systems, IAM facilitates communication between applications running on different computers and operating systems. SwitchKit also provides programmability for intelligent channel allocations, which allow applications to select appropriate outgoing ports. Typically, an application must manage channel idle/busy states and implement

intelligence to route them. SwitchKit manages this process across all applications, allowing you to choose from a number of selection schemes, including the following:

- Round Robin
- Least Recently Used
- Most Recently Used
- Ascending/Descending

Intelligent Channel Allocation

Channels are allocated to applications upon demand, based on the most available resource in the requested group or set of groups. Intelligent channel allocations can be used between and across applications, because they may be sharing the same trunks. SwitchKit also load shares messaging and channel allotments across applications, based on a programmable load factor that is under application control. Conversely, you can program the system not to load share if another scheme is more appropriate.

Internal Messages

Some functions shown in the API function header files are for internal use only. Among those functions are:

- sk_openFile()
- sk_getSwitchmgrId()
- sk_downloadFile()
- sk_newConfig()

Using SwitchKit API

Overview

SwitchKit application developers require the SwitchKit API to allow their application to connect to the LLC. Sample applications are provided for guidance and review. Additionally, UNIX makefiles and Microsoft Visual Studio® project workspaces are included in the installation. These examples provide information on necessary libraries that are required for all SwitchKit applications.

For the UNIX operating environments, SwitchKit API is available in two sets of libraries: standard SK API and threadsafe SK API. Both APIs offer the same basic functionality to the application developer, with one major difference. The standard SK API allows single-threaded applications to be developed to connect to the LLC. The threadsafe SK API allows multi-threaded applications to be developed to connect to LLC. For maximum flexibility in application functionality today and in the future, it is recommended that all new development occur on the threadsafe SK API.

All SK API libraries are found in the *SK_LIB_DIR\lib* directory. See *Development Libraries (1-5)*.

Solaris

If you are using the Solaris operating systems, you have the option of using the Forte compiler available through Sun Microsystems, or GNU gcc version 3.4.3. You also have the option of using the standard version of SK API or the threadsafe version of the SK API. SwitchKit API is available to support these situations.

Linux

If you are using Red Hat Linux operating systems, you have the option of using the threadsafe library or not. SwitchKit API is available to support both.

HP-UX

If you are using the HP-UX operating system, you have the option of using the threadsafe library or not. SwitchKit API is available to support both. Whether you are developing using the threadsafe or standard libraries, you must use *libskcommonAA.sl.*

Development Libraries

The next table shows which libraries are required for development on each UNIX platform supported. Be certain to include all libraries in your makefile for proper SwitchKit application development.

Operating System	Libraries
Windows®	***Non Threadsafe Development*** skapi.lib ***Threadsafe Development*** skapi_ts.lib ACE.lib
Solaris	***Non Threadsafe Development*** libskapi.a libskcapi.a libskapiGCC.a libskcapiGCC.a ***Threadsafe Development*** libskapi_ts.a libskcapi_ts.a libskapiGCC_ts.a libskcapiGCC_ts.a lib/libACE.so.5.4.4 libACE.so (soft link to libACE.so.5.4.4)
Linux	***Non Threadsafe Development*** libskapi.a libskcapi.a ***Threadsafe Development*** libskapi_ts.a libskcapi_ts.a lib/libACE.so.5.3.3 libACE.so (soft link to libACE.so.5.3.3)

Operating System	Libraries
HP-UX	***Non Threadsafe Development*** libskapiAA.sl libskapi.sl (soft link to libskapiAA.sl) libskcapiAA.sl libskcapi.sl (soft link to libskcapiAA.sl) *libskcommonAA.sl *libskcommon.sl (soft link to libskcommonAA.sl) *libstateAA.sl (required for LLC functionality) *libstate.sl (soft link to libstateAA.sl) ***Threadsafe Development*** libskapiAA_ts.sl libskcapiAA_ts.a libskcommonAA_ts.sl *libskcommonAA.sl *libskcommon.sl (soft link to libskcommonAA.sl) libACE.sl

* *libskcommon* is required for proper operation of LLC, SwitchManager, and customer-developed applications. *libstate* is required for proper operation of LLC.

Microsoft Windows NT® and Windows XP

If you are using a Windows NT® or WIndows XP operating system to develop your application, the following files must be included in your workspace.

Development Library	SKApi.lib
Run-time Library	SKApi.dll

API Messages Used for Call Control Programming

Overview This section outlines the messages used for developing call control applications.

Call Control

BusyOut
CallProcessingEvent
CPAResult
ChannelReleased
ChannelReleasedWithData
ChannelReleaseRequest
CollectDigitString
ConferenceCreate
ConferenceDeleted
ConferenceDeleteRequest
Connect
ConnectOneWayForced
ConnectOneWayToConference
ConnectToConference
ConnectTonePattern
ConnectWait
ConnectWithData
ConnectWithPad
CPCDetection
CrossConnectChannel
CrossConnectSpan
CrossDisconnectChannel
CrossDisconnectSpan
DisconnectTonePattern

DS0StatusChange
DSPCacheModify
DSPServiceCancel
DSPServiceRequest
GenerateCallProcessingEvent
InseizeControl
OutpulseDigits
OutseizeControl
ParkChannel
PlayFileModify
PlayFileStart
PlayFileStop
PPLEventIndication
PPLEventRequest
RecAnnConnect
RecAnnDelete
RecAnnDisconnect
RecAnnDownload
RecAnnDownloadInit
RecordFileModify
RecAnnFSConvert
RecAnnFSDefrag
RecAnnSingleDelete
RecordFileModify
RecordFileStart
RecordFileStop
ReleaseChannel
ReleaseWithData
Remove Channels From Group

Request For Service
ResourceConnect
Resource Create
ResourceDelete
Resource Modify
RFSWithData
Route Control

2 Connection Management

Purpose This chapter describes the connection management functions available in SwitchKit, as well as the SwitchKit API messages for application connection management.The chapter also provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). A non-zero return value indicates some error condition. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

Introduction to Connection Management

Description

In a SwitchKit environment, an application communicates with one or multiple LLCs. SwitchKit provides a set of functions that help you create and manage connections between your application module and any LLC your application needs to communicate with.

There are three models of connecting applications to the LLC:

- Applications Connecting to Multiple LLCs

  Use this model if you are writing a new application that connects to multiple LLCs. In this context, a redundant pair of LLCs is considered a single LLC.

- Connecting Legacy Applications to Multiple LLCs

  This model is for legacy applications (before SwitchKit version 5.7) connecting to multiple LLCs. Do not use this model if you are writing a new application.

- Applications Connecting to One LLC

  Use this model if you are writing a new application that connects to one LLC, or if you have an existing application connecting to one LLC. In this context, a redundant pair of LLCs is considered a single LLC.

Consider your system's current and future needs when deciding to use the functions offered either under Applications Connecting to Multiple LLCs, or under Applications Connecting to One LLC, for your new application development.

Applications Connecting to Multiple LLCs

Description

This model is for an SwitchKit environment with multiple LLCs. Use this model if you are writing a new application that must connect to multiple LLCs.

In this model, your application specifies the connection information in the create connection function call and assigns a connection ID and an application name to that connection. From then on, all references to the connection are made using the connection ID. This eliminates any ambiguity about the LLC connection. The connection ID is returned with each event, which makes it simple to determine which LLC should receive subsequent action.

How to use the Connection Model with ID

To connect to multiple LLCs and to manage those connections, your application must call SK_Connection ***sk_createConnectionWithID***(). With that function call, an integer connection ID is assigned to the connection between the application and that particular LLC.

When the connection ID is assigned, your application must call one of the ***sk_*OnConnection***() functions to interact with a particular LLC. The ***sk_*OnConnection***() function takes the assigned connection ID as a parameter.

Functions in this Model

Following is a list of functions that are to be used in this model:

```
SK_Connection *sk_createConnectionWithID();
sk_*OnConnection();
int sk_closeNamedConnection();
int sk_getLLCSocketDescriptorForConnection();
int sk_getConnectionNameForConnection();
int sk_getConnectionForID;
int sk_destroyConnectionForced();
int sk_appDescriptionData();
```

SK_Connection*

SK_Connection is an opaque object which is returned by several sk_createConnection*() functions that can be used to refer to a specific connection in other SwitchKit connection management functions such as ***sk_getSpecificConnectionName()***.

sk_createConnectionWithID()

Description Use the ***sk_createConnectionWithID***() function to create a connection between your application module and a specific LLC.

> **Important!** If you are using the function sk_appDescriptionData(), the aLabel argument in sk_createConnectionWithID() matches anAppName in sk_appDescriptionData().

Syntax

```
SK_Connection *sk_createConnectionWithID(int aConid, const
   char *aLabel, int anAppID, int isForcedFlag, const
   char *aPriHost, int aPriPort, const char *aBackupHost,
   int aBackupPort);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aConID	aConID is a connection identifier specified by the application and used to refer to an applications connection to a specific LLC. The connection identifier is used by the SwitchKit API to route messages from the application to a specific LLC. The connection identifier is initially defined for the API when the application calls sk_createConnectionWithID() or skts_createConnection() and should be used for any sk_*OnConnection() and skts_*() functions where required.
aLabel	aLabel is a text string used by the application to identify the connection. The value has no meaning to the SwitchKit API and is available for query via sk_getSpecificConnectionLabel() and skts_getSpecificConnectionLabel().
anAppID	anAppID is the application identifier used by the LLC to route messages to the application and to log any events of interest regarding the application. If the caller of the function does not care which application identifier it receives, it can specify -1. This allows LLC to select the next available application identifier beginning at 10. If the specified application identifier is already used, the LLC will assign the next available identifier beginning at the requested identifier, unless the isForcedFlag is set to 1 or sk_initializeConnectionForced() was used to establish the connection with LLC.
isForcedFlag	If isForcedFlag equals 1, then the application identifier for this application will be set to the value specified in anAppID even if that application identifier is being used by another application.
aPriHost	The IP address of the primary LLC with the following format, for example: “10.10.55.10”
aPriPort	The IP port number the primary LLC the primary LLC is listening to.

Argument	Description
aBackupHost	The IP address of the secondary LLC, in the following format, for example: “10.10.55.10”. If there is no secondary LLC, this field should be set to "". That is, the 0-length string.
aBackup Port	The IP port for the secondary LLC is listening to.

Return Values

This is one of the functions that returns an integer value pointer of an integer. A `NULL` pointer indicates an error condition.

Useful Related Functions

```
sk_*OnConnection();
int sk_closeNamedConnection();
int sk_getLLCSocketDescriptorForConnection();
int sk_getConnectionNameForConnection();
int sk_getConnectionForID;
int sk_destroyConnectionForced();
int sk_appDescriptionData();
```

sk_*OnConnection()

Functions using OnConnection

Some SwitchKit functions are duplicated as “OnConnection” functions. For example, ***sk_watchChannelGroup()*** is also available as ***sk_watchChannelGroupOnConnection().***

The difference with the ***sk_*OnConnection***() functions is the integer connection parameter:

```
int aConID
```

However, if ***sk_createConnectionWithID***() was used to establish a connection between the LLC and the application, the functions with OnConnection must be used to create interaction between the LLC and the application. The following is a list of functions used that have with.

On Connection Functions

Channel Management Functions

For more information on these functions, see *Channel Management (3-1)*.

sk_associateChannelGroupOnConnection
sk_broadcastLoadOnConnection
sk_clearChannelGroupOnConnection
sk_configChannelGroupOnConnection
sk_ignoreChannelGroupOnConnection
sk_monitorChannelOnConnection
sk_removeChannelsFromGroupOnConnection
sk_requestChannelOnConnection
sk_requestOutseizedChannelOnConnection
sk_requestRouteControlledChannelOnConnection
sk_returnChannelOnConnection
sk_unMonitorChannelOnConnection
sk_watchChannelGroupOnConnection

Register Functions

For more information on these functions, see *Register Functions (4-1)*.

sk_appGroupRegisterOnConnection
sk_msgRegisterOnConnection
sk_msgUnRegisterOnConnection
sk_pplComponentRegisterOnConnection
sk_pplComponentUnRegisterOnConnection
sk_pplTCAPRegisterOnConnection
sk_pplTCAPUnRegisterOnConnection

Logging Functions

For more information on these functions, see *Logging Functions (5-1)*.

sk_logLevelOnConnection
sk_logMessageOnConnection

Message Functions

For more information on these functions, see *Message Functions (7-1)*.

sk_rcvAndDispatchOnConnection
sk_rcvAndDispatchAutoStorageOnConnection
sk_rcvMessageOnConnection
sk_sendMessageOnConnection
sk_sendMessageWithHandlerOnConnection
sk_sendMessageWithTagOnConnection
sk_sendMsgStructOnConnection

Redundancy Functions

For more information on these functions, see *Redundancy (10-1)*.

sk_activateExnetMatrixOnConnection
sk_deregisterAsRedundantAppOnConnection
sk_registerAsRedundantAppOnConnection

Additional Functions to Connect to Multiple LLCs

sk_closeNamed Connection()

This function closes the connection with the specified connection ID between the application and the LLC.

Syntax

```
int sk_closeNamedConnection( int aConID);
```

How closeConnection works

The LLC no longer has a socket open to this application. However, the ConnectionManager object remains and causes the connection to re-establish in case of subsequent calls to ***sk_rcvAndDispatch***() or attempts to send a message.

sk_getConnectionNameFor Connection()

The function call ***sk_getConnectionNameForConnection***() uses the connection ID to retrieve the connection name of the connection between the applications and the LLC.

Syntax

```
int sk_getConnectionNameForConnection( int aConID);
```

How getConnectionName works

You can call this function only after opening a socket to the LLC. To open the socket, send a harmless message through that connection.

sk_getLLCSocket DescriptorForConnection()

The function ***sk_getLLCSocketDescriptorForConnection***() returns the socket descriptor associated with the current application. You need this function if your application is connected to the LLC, to a database, or to another server.

Syntax

```
int sk_getLLCSocketDescriptorForConnection( int aConID);
```

Threadsafe Syntax

```
int skts_getLLCSocketDescriptor(int aConID);
```

How getLLCSocketDescriptor works

By calling ***sk_getLLCSocketDescriptorForConnection***() your application retrieves the socket descriptor associated with the specified connection ID from the LLC. This enables your application to issue an OS-specific call, for example ***select()***. This indicates when there is

activity on the LLC socket and then the application must call the function ***sk_rcvAndDispatch***() to begin processing of all messages on the LLC socket.

The ***sk_getLLCSocketDescriptorForConnection***() function should be called each time ***select()*** is called, because the socket descriptor can change without the application being aware of it.

For example, if the PLLC fails and the RLLC becomes active, the socket of the PLLC will no longer be valid. A subsequent call to the function provides the socket descriptor for the now active RLLC.

sk_getConnectionForID()

The function call ***sk_getConnectionForID***() retrieves the SK_Connection object for the specified connection ID.

Syntax

```
SK_Connection *sk_getConnectionForID(int aConID);
```

sk_destroyConnection Forced()

Your application should use the function ***sk_destroyConnectionForced***() to destroy a connection to the LLC with the specified connection ID. If the specified connection is the last connection opened, it will be destroyed but cause a return value `SK_NO_CONNECT_AVAIL`. As the name implies, this return indicates that no connections remain. However, should your application call ***sk_rcvAndDispatch***(), the system attempts to establish a connection to an LLC using the default connection arguments specified in the SK_LLC_Host/Port environment variables, or at local host:1312.

This function call frees up all the memory allocated in ***sk_createConnectionWithID***().

Syntax

```
int sk_destroyConnectionForced(int aConID);
```

Threadsafe Syntax

```
int skts_destroyConnectionForced(int aConID);
```

Return Values

`SK_NO_CONNECT_AVAIL`

This return indicates that all connections have been destroyed.

Connecting Legacy Applications to Multiple LLCs

Description

Do not use this model if you intend to write a new application. This model is only to be used by existing appliactions, written to run on SwitchKit versions before 5.7.

Functions in this category rely heavily on the SK_Connection pointer to determine or provide information and context. SK_Connection is returned when the connection to the LLC is first established. If your application must make a request to the LLC or needs to receive a message, if would first have to send function calls to get or set the current connection to the specific SK_Connection pointer.

How to use the Connection Model without ID

To connect to multiple LLCs and to manage those connections, your application must call SK_Connection ****sk_createConnection***(). Every call creates a separate SK_Connection pointer for each connection. To send a message to a specific LLC, your application must call the function ***sk_setCurrentConnection***() with the SK_Connection pointer at the desired LLC. Without that function call and specifying pointer, there is no way to direct a message to an LLC other than the one for which SwitchKit is processing the current message.

Functions in this Model

```
SK_Connection *sk_createConnection();
int sk_getLLCSocketDescriptor();
int sk_getSpecificConnectionName();
const char *sk_getSpecificConnectionLabel();
SK_Connection *sk_getCurrentConnection();
void sk_setCurrentConnection();
int sk_destroyConnection();
int sk_appDescriptionData();
```

sk_createConnection()

Description Use the SK_Connection ***sk_createConnection***() function to create a connection between your application module and a specific LLC.

> **Important!** If you are using the function sk_appDescriptionData(), the aLabel argument in sk_createConnection() matches anAppName in sk_appDescriptionData().

Syntax

```
SK_Connection *sk_createConnection(const char *aLabel, int
   anAppID, int isForcedFlag, const char *aPriHost, int
   aPriPort, const char *aBackupHost, int aBackupPort);
int *skts_createConnection(int aConID, const char *aLabel,
   int anAppID, int isForcedFlag, const char *aPriHost,
   int aPriPort, const char *aBackupHost, int
   aBackupPort);
```

Parameters The function parameters are shown in the next table.

Argument	Description
aConID	aConID is a connection identifier specified by the application and used to refer to an application's connection to a specific LLC. The connection identifier is used by the SwitchKit API to route messages from the application to a specific LLC. The connection identifier is initially defined for the API when the application calls sk_createConnectionWithID() or skts_createConnection() and should be used for any sk_*OnConnection() and skts_*() functions where required.
aLabel	aLabel is a text string used by the application to identify the connection. The value has no meaning to the SwitchKit API and is available for query via sk_getSpecificConnectionLabel() and skts_getSpecificConnectionLabel().
anAppID	anAppID is the application identifier used by the LLC to route messages to the application and to log any events of interest regarding the application. If the caller of the function does not care which application identifier it receives, it can specify -1. This allows LLC to select the next available application identifier beginning at 10. If the specified application identifier is already used, the LLC will assign the next available identifier beginning at the requested identifier, unless the isForcedFlag is set to 1 or sk_initializeConnectionForced() was used to establish the connection with LLC.
isForcedFlag	If isForcedFlag equals 1, then the application identifier for this application will be set to the value specified in anAppID even if that application identifier is being used by another application.
aPriHost	The IP address of the primary LLC with the following format, for example: "10.10.55.10"

Argument	Description
aPriPort	The IP port number the primary LLC the primary LLC is listening to.
aBackupHost	The IP address of the secondary LLC, in the following format, for example: “10.10.55.10”. If there is no secondary LLC, this field should be set to "". That is, the 0-length string.
aBackup Port	The IP port for the secondary LLC is listening to.

Return Values

sk_createConnection() is one of the functions that returns an integer value instead of a pointer. A `NULL` pointer indicates an error condition.

****skts_createConnection() returns***- returns the following values:

aConID if connection successful or previously created

SK_ERROR_CREATING_CONNECTION (negative integer) if connection fails

Useful Related Functions

```
int sk_getLLCSocketDescriptor();
int sk_getSpecificConnectionName();
const char *sk_getSpecificConnectionLabel();
SK_Connection *sk_getCurrentConnection();
void sk_setCurrentConnection();
int sk_destroyConnection();
int sk_appDescriptionData();
```

sk_createConnectionWithID()

Description Use the SK_Connection ***sk_createConnectionWithID***() function to create a connection between your application module and a specific LLC.

Syntax

```
SK_Connection *sk_createConnectionWithID(int aConID, const
   char *aLabel, int anAppID, int isForcedFlag, const
   char *aPriHost, int aPriPort, const char aBackupHost,
   int aBackupPort);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aConID	The user-defined unique connection ID to a particular LLC.
aLabel	A user-defined text string to identify the connection, but it has no meaning to the connection management code.
anAppID	anAppID is the application identifier used by the LLC to route messages to the application and to log any events of interest regarding the application. If the caller of the function does not care which application identifier it receives, it can specify -1. This allows LLC to select the next available application identifier beginning at 10. If the specified application identifier is already used, the LLC will assign the next available identifier beginning at the requested identifier, unless the isForcedFlag is set to 1 or sk_initializeConnectionForced() was used to establish the connection with LLC.
isForcedFlag	If isForcedFlag equals 1, then the application identifier for this application will be set to the value specified in anAppID even if that application identifier is being used by another application.
aPriHost	The IP address of the primary LLC with the following format, for example: “10.10.55.10”
aPriPort	The IP port number the primary LLC the primary LLC is listening to.
aBackupHost	The IP address of the secondary LLC, in the following format, for example: “10.10.55.10”. If there is no secondary LLC, this field should be set to "". That is, the 0-length string.
aBackup Port	The IP port for the secondary LLC is listening to.

Return Values

This is one of the functions, that returns an integer value pointer of an integer. A `NULL` pointer indicates an error condition.

Useful Related Functions

```
sk_*OnConnection();
int sk_closeNamedConnection();
int sk_getLLCSocketDescriptorForConnection();
int sk_getConnectionNameForConnection();
```

```
int sk_getConnectionForID;
int sk_destroyConnectionForced();
int sk_appDescriptionData();
```

Additional Functions for Legacy Applications

sk_getSpecificConnectionName()

With the function ***sk_getSpecificConnectionName***() your application can retrieve the numeric name of that connection. This function does not change the current connection.

Syntax

```
int sk_getSpecificConnectionName(SK_Connection
   *aConnection);
```

Threadsafe Syntax

```
int sk_getSpecificConnectionName(int aConID);
```

sk_getSpecificConnectionID()

With the function ***sk_getSpecificConnectionID***() your application can retrieve the connection ID of that connection. This function does not change the current connection.

Syntax

```
int sk_getSpecificConnectionID(SK_Connection *aConnection);
```

sk_getSpecificConnectionLabel()

The function ***sk_getSpecificConnectionLable***() returns the label associated with a connection that was created with ***sk_createConnection***().

Syntax

```
int sk_getSpecificConnectionLabel(SK_Connection
   *aConnection);
```

sk_getCurrentConnection()

If your application is connected to more than one LLC, the LLC receiving a message or function call makes that connection current. Unless otherwise specified, the LLC communicates with your application through that current connection.

The function SK_Connection ****sk_getCurrentConnection***() enables another process of your application to get the current connection.

Syntax

```
int SK_Connection *sk_getCurrentConnection();
```

sk_setCurrentConnection()

Your application should issue the function call ***sk_setCurrentConnection***() to make that connection current. Any future communication goes through that connection

Syntax

```
int sk_setCurrentConnection(SK_connection *aConnection);
```

sk_destroyConnection()

Your application should use the function ***sk_destroyConnection***() to destroy a connection to the LLC. You cannot use this function on your current connection. Therefore, this function can destroy all connections between applications and the LLC, except the last connection left. This function call frees up all the memory allocated in ***sk_createConnection***().

Syntax

```
int sk_destroyConnection(SK_Connection *aConnection);
```

Threadsafe Syntax

```
int skts_destroyConnection( int aConID );
```

Return Values

SK_CONNECT_IN_USE	This return value indicates that you called the function on the current connection.

Applications Connecting to One LLC

Description

If you are working in an environment with only one LLC and if you are not going to change to an environment with multiple LLCs, you can use this model.

How to use the Simplified Connection Model

To connect to just one LLC in the system, your application can call ***sk_initializeConnection***() or ***sk_initializeForcedConnection***(). These function calls initialize a socket to the LLC with a number named connectionName.

Calling these functions does not open the socket. The socket must be opened by sending a message with the specific connectionName to the LLC.

How initializeConnection works

If your application calls ***sk_initializeConnection***(), any previous connection to the LLC is terminated, and a new one is initialized with the specified connection name.

If a requested connection name is already in use, the first available integer greater than connectionName is assigned automatically to the application. To ensure that your application receives an unused connection name, call the function with an argument of -1.

How initializeForcedConnection works

The function ***sk_initializeForcedConnection***() behaves exactly as ***sk_intializeConnection***() does, except when connectionName is already in use by another application to the LLC. Instead of receiving a new, unused connection name, your application insisting on the specified name causes the first connection to terminate.

Functions in this Model

```
int sk_initializeConnection();
int sk_initializeForcedConnection();
int sk_closeConnection();
int sk_getLLCSocketDescriptor();
int sk_getConnectionName();
```

sk_initializeConnection() / sk_initializeForcedConnection()

Description

Use a function call to ***sk_initializeConnection***() or ***sk_initializedForcedConnection***() function to establish a socket between your application and the LLC. The connection will be established with the IP and port numbers specified in the environment variables SK_LLC_HOST/PORT and SK_RLLC_HOST/PORT.

Syntax

```
int sk_initializeConnection(int anAppID);
int sk_initializeForcedConnection(int anAppID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
anAppID	anAppID is the application identifier used by the LLC to route messages to the application and to log any events of interest regarding the application. If the caller of the function does not care which application identifier it receives, it can specify -1. This allows LLC to select the next available application identifier beginning at 10. If the specified application identifier is already used, the LLC will assign the next available identifier beginning at the requested identifier, unless the isForcedFlag is set to 1 or sk_initializeConnectionForced() was used to establish the connection with LLC.

Return Values

Possible return values for this function:

SK_CONNECT_IN_USE — This return value indicates that the specified connection name is already in use and that the connection opened with another name, which can be requested with the function call ***sk_getConnectionName***.

Useful Related Functions

```
int sk_getConnectionName();
int sk_closeConnection();
int sk_getLLCSocketDescriptor();
```

Additional Functions used with One LLC

sk_closeConnection()

The function closes the current connection between the application and the LLC.

Syntax

```
int sk_closeConnection();
```

How closeConnection works

The LLC no longer has a socket open to this application. However, the ConnectionManager object remains and causes the connection to re-establish in case of subsequent calls to ***sk_rcvAndDispatch***() or attempts to send a message.

sk_getConnectionName()

The function call ***sk_getConnectionName***() retrieves the name registered to the applications current connection between the application and the LLC.

Syntax

```
int sk_getConnectionName();
```

How getConnectionName works

You can call this function only after opening a socket to the LLC. To open the socket, you should send a harmless message through that connection.

sk_getLLCSocket Descriptor()

The function ***sk_getLLCSocketDescriptor***() returns the socket descriptor associated with the current application. You need this function if your application is connected to the LLC, to a database, or to another server.

Syntax

```
int sk_getLLCSocketDescriptor();
```

How getLLCSocketDescriptor works

By calling ***sk_getLLCSocketDescriptor***(), your application retrieves the socket descriptor from the LLC, which enables your application to issue an OS-specific call, for example ***select***(). This indicates when there is activity on the LLC socket and only then calls the function ***sk_rcvAndDispatch***().

The ***sk_getLLCSocketDescriptor***() function should be called each time select() is called, because the socket descriptor can change without the knowledge of the application.

For example, if the PLLC fails and the RLLC becomes active, the socket of the PLLC will no longer be valid. A subsequent call to the function provides the socket descriptor for the now active RLLC.

Functions used in all Connection Models

sk_appDescriptionData()

To provide additional information uniquely identifying your application, use the ***sk_appDescriptionData()*** function. This function should be called after your application has established a connection to the LLC. This function is also available as ***skts_appDescriptionData()***.

Important! If you are using the functions sk_createConnection() or sk_createConnectionWithID(), the argument anAppName matches the aLabel argument.

Syntax

```
void sk_appDescriptionData(char *anAppName,char
   *aShortName, char *anAppVersion, char *aUserData)
void skts_appDescriptionData(char *anAppName,char
   *aShortName, char *anAppVersion, char *aUserData);
```

Parameters

These are the function parameters.

Arguments	Description
anAppName	This is meant to be a descriptive name for the application.
aShortName	This is used to name the *maintenance.log* log file. It is not passed to the LLC.
AnAppVersion	This string should describe the version of the application software.
aUserData	This is stored by the SwitchKit API, but it is used only by the application.

APIs used in all Connection Models

List of SwitchKit APIs

The API messaged listed below are used in connection models. These API messages are found in the *API Reference*.

- AppConnectionQuery
- AppPopulationQuery
- Connect 0x0000
- Card Population Query 0x0007

3 Channel Management

Purpose

This chapter describes the channel management functions available in SwitchKit, as well as a list of the SwitchKit API messages used for channel management. The chapter also provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

Your application modules will notify the LLC of which channels to use, using the Channel Management functions. Most of these functions are based on channel groups. A channel group is a text string that is associated with a set of channels. The association between the group name and channels is specified in the configuration. Inbound and outbound channel groups are managed differently.

Channels

Inbound Channels

In order to receive inbound calls, your application must notify the LLC that it is willing to handle messages that arrive on inbound channels. You do not need to send messages to those channels until the application is notified of an inbound call. When a call arrives on an inbound channel, the LLC can route it to any application willing to handle that channel.

Thereafter, the application is able to send messages to that channel and to receive all messages concerning that channel. Through the ***sk_watchChannelGroup***() function, the application notifies the LLC that it is willing to handle a channel group. If multiple applications watch the same channel group, load sharing occurs.

For example, if you create a channel group “All_Inbound” in your configuration file to receive incoming calls, your application must issue the following function call:

sk_watchChannelGroup(“All_Inbound”);

This function call tells the LLC that any calls on the channels associated with the group “All_Inbound” can be routed to your application.

The messages that are sent to your application include information about the channel they arrive on. Your application does not need to keep track of specific channels in the group it is watching. However, it is possible for the application to obtain specific information about the channels through other API messages.

Outbound Channels

Unlike inbound channels, your application must initiate the communication with outbound channels, rather than waiting for a message to arrive. Therefore, to obtain access to an outbound channel, your application must explicitly request a channel from an outbound channel group, which is usually done simultaneously with an outseizure. To do this use ***sk_requestOutseizedChannel()***.

Syntax for sk_requestOutseizedChannel()

```
int sk_requestOutseizedChannel (char *aChannelGroup, int
   aNumRetries, XL_OutseizeControl *anOutseizeMsg, void
   *aTag, HandlerFunc *aHandlerFunc);
```

Intelligent Channel Allocation

When your application requests a channel from a channel group, the LLC determines which channel to allocate, based on an intelligent channel allocation algorithm. The allocation is based on which subgroup of the requested channel group is most available.

If the switch has outbound trunks to multiple inter-exchange carriers, you can define one overall outbound channel group, called “outbound” that incorporates all outbound trunks. You can then define specific groups to Carrier A and Carrier B, called “outboundA” and “outboundB.” The “outboundA” and “outboundB” groups are subgroups of the “outbound” group.

If an application requests an outseize on a port to a specific carrier, LLC finds an unused channel in the subgroup for that specific carrier that it can successfully outseize on, and provides the outseized channel to the application module.

However, if an application does not care which specific carrier it is outseized to, and requests a channel from group “outbound,” the LLC determines which of “outboundA” or “outboundB” is less loaded, and returns a successfully outseized channel from that subgroup. The LLC always attempts to preserve the most limited channel group for the applications that require it.

> **Important!** Channels can only be managed by either the CSP or SwitchKit. Do not attempt to use LLC channel management when using CSP routing.

Application Disconnection Functionality

When an application terminates, you can specify how the LLC handles the non-idle channels using the environment variable: SK_CHANNEL_RECOVERY_METHOD. If this variable is set to 0x01, channels will be released. If this variable is set to 0x02, channels will be taken out-of-service. If this variable is set to 0x00, no action is taken. If this variable is set to 0x03, all channels will be taken out-of-service within a group.

Channel Groups

In channel management, if an application specifies a channel group that was not configured in LLC using an *AssociateChannelGroup* API message, the LLC will create that group and assign no channels to the channel group. When *AssociateChannelGroup* is sent to the LLC, LLC will add the specified channels to that group.

If LLC receives a reference to a channel group that does not currently exist, it will make note of that fact in the maintenance.llc.log file. LLC also sets up this invalid channel group with a group name and zero channels. If an *AssociateChannelGroup* is sent later, that invalid channel group will get populated.

SK_AllAssignedChannels

Description

The *SK_AllAssignedChannels* channel group is a pseudo-channel group that can be watched by applications, but cannot have channels associated with it. It is a channel group that, in essence, consists of all channels, but if queried, contains no channels.

Functionality with Request For Service

If a *RequestForService* or *RFSWithData* is received by the LLC from the switch indicating that a call has arrived for a particular channel, and there is no application watching the channel group(s) that the channel belongs to, the behavior of the LLC with regards to the *RequestForService* or *RFSWithData* messages depends on whether an application is watching the SK_AllAssignedChannels channel group.

If one or more applications are watching the SK_AllAssignedChannels channel group when the above situation occurs, one of the applications watching the channel group will be presented the *RequestForService* or *RFSWithData*. The selection of application takes place in the same manner as any other application selection when a channel group is being watched by multiple applications.

If no applications are watching the SK_AllAssignedChannels channel group, the LLC will discard the message and optionally log that fact (based on the value of the SK_WARN_ORPHAN_RFS environment variable).

Important! There are two significant differences between the *SK_AllAssignedChannels* channel group and other channel groups.

- You cannot perform an *AssociateChanGroup* on this channel group.
- Calls will only be made available to the application watching this channel group as a last resort (only if no applications are watching any other channel groups the channel is associated with).

sk_associateChannelGroup()

Description

The ***sk_associateChannelGroup***() function is used to manage the physical channels mapping to logical channel groups. When the automatic configuration facilities in EXS SwitchManager are used, this function is not needed. If a group does not exist, ***sk_associateChannel Group***() automatically creates the group. The function call is cumulative, so if you want to build a non-contiguous set of channels, just issue multiple ***sk_associateChannelGroup***() function calls. This function is also available as ***sk_associateChannelGroupOnConnection()*** and ***skts_associateChannelGroup()***.

Syntax

```
int sk_associateChannelGroup(char *aChannelGroup, int
   aStartSpan, int aStartChannel, int anEndSpan, int
   anEndChannel);
int sk_associateChannelGroupOnConnection(char
   *aChannelGroup, int aStartSpan, int aStartChannel, int
   anEndSpan, int anEndChannel, int aConID);
```

Threadsafe Syntax

```
int skts_associateChannelGroupOnConnection(char
   *aChannelGroup, int aStartSpan, int aStartChannel, int
   anEndSpan, int anEndChannel, int aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Identifies the channel group.
aStartSpan	Identifies the first span in the group.
aStartChannel	Identifies the first chan in the group.
anEndSpan	Identifies the last span in the group.
anEndChannel	Identifies the last chan in the group.
aConID	A connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_broadcastLoad()

Description

Your application modules can update their load with the LLC by a call to ***sk_broadcastLoad***(). The load tells the LLC how busy the process is. High values are busier than low values.

If the processes do not call ***sk_broadcastLoad***(), then the LLC distributes the messages in an alternating manner between all available application modules.

This function is also available as ***sk_broadcastLoadOnConnection()*** and ***skts_broadcastLoad()***.

Syntax

```
int sk_broadcastLoad(double aLoadFactor);
int sk_broadcastLoadOnConnection(double aLoadFactor, int
   aConID);
```

Threadsafe Syntax

```
int skts_broadcastLoad(double aLoadFactor, int aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aLoadFactor	The load value changes the load distribution.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_clearChannelGroup()

Description With the function call to ***sk_clearChannelGroup()***, LLC removes all channels from the specified group and removes the group. After calling this function, if a request channel was sent for the group removed, LLC would return a response of "invalid group". Use the ***sk_removeChannelsFromGroup()*** function to remove individual channels from a channel group. The ***sk_clearChannelGroup***()function is also available as ***sk_clearChannelGroupOnConnection()*** and ***skts_clearChannelGroup()***.

Syntax

```
int sk_clearChannelGroup(char *group);
int sk_clearChannelGroupOn Connection(char *group, int
   aConID);
int skts_clearChannelGroup(char *aChannelGroup, int
   aConID );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Identifies the channel group.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_configChannelGroup()

Description The function ***sk_configChannelGroup***() allows you to configure the channel allocation in the specified channel group. This function is also available as ***sk_configChannelGroupOnConnection()*** and ***skts_configChannelGroup***.

Syntax

```
int sk_configChannelGroup(char *aChannelGroup, int
   anEntity, int aParameter);
int sk_configChannelGroupOnConnection(char
   *aChannelGroup, int anEntity, int aParameter, int
   aConID);
int skts_configChannelGroup( char *aChannelGroup, int
   anEntity, int aParameter, int aConID );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Specifies the channel group.
anEntity	Specifies the attribute to be configured. Only one is currently defined, which is **Entity_Alloc**, the channel allocation pattern.
aParameter	Defines the method of channel allocation: • **Alloc_Round Robin** • **Alloc_Ascending** • **Alloc_Descending** • **Alloc_LRU** (Last Recently Used) • **Alloc_MRU** (Most Recently Used)
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

SK_BAD_PARAM	When entity is not zero or parameter is not between 0 and 4 (inclusive.)

sk_getAssignedChannelGroup()

Description The function ***sk_getAssignedChannelGroup***() allows your application to determine the channel group that a channel has been assigned to. This function returns the channel group that was specified in the call to ***sk_watchChannelGroup***() that caused the LLC to assign Span and Channel to this application. For channels that were allocated to the application through a different mechanism, such as through ***sk_requestOutseizedChannel***(), this function returns NULL.

Syntax

```
char *sk_getAssignedChannelGroup(int aSpan, int
   aChannel);
char * skts_getAssignedChannelGroup(int aSpan, int
   aChannel);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	Specifies the Span.
aChannel	Specifies the Channel.

sk_ignoreChannelGroup()

Description The ***sk_ignoreChannelGroup***() function stops the LLC from routing messages on new channels of the specified group to the application issuing the function call. Any channels of calls in progress will still be registered to the application. This function is also available as ***sk_ignoreChannelGroupOnConnection()*** and ***skts_ignoreChannelGroup()***.

Syntax

```
int sk_ignoreChannelGroup(char *aChannelGroup);
int sk_ignoreChannelGroupOnConnection(char
   *aChannelGroup, int aConID);
int skts_ignoreChannelGroup( char *aChannelGroup, int
   aConID );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Identifies the channel group.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_monitorChannel() / sk_unMonitorChannel()

Description

Use ***sk_monitorChannel***() to monitor the messages that occur on a specific channel. This function is also available as ***sk_monitorChannelOnConnection()*** and ***skts_monitorChannel()***.

Your application can monitor a channel whether or not the channel is associated with another application. Monitoring a channel has no effect on the automatic channel allocation process. The ***sk_requestOutseizedChannel***() function can still return a channel that is being monitored.

Monitoring a channel causes a copy of outgoing messages to the switch and incoming messages from the switch to be routed to the monitoring application. Outgoing messages are only routed to the monitoring application when the LLC gets its acknowledgment. The sequence number is then filled in with the status of that acknowledgment. If a monitoring application gets an outseize control message, it can look at its sequence number to see the status of the acknowledgment to that outseize control message. Acknowledgement messages are not sent to monitoring applications.

Use ***sk_unMonitorChannel***() to stop monitoring the messages that occur on a specific channel. This function is also available as ***sk_unMonitorChannelOnConnection()*** and ***skts_unMonitorChannel()***.

Syntax

```
int sk_monitorChannel(int aStartSpan, int aStartChannel,
   int anEndSpan, int anEndChannel);
int sk_unMonitorChannel(int aStartSpan, int
   aStartChannel, int anEndSpan, int anEndChannel);
int sk_monitorChannelOnConnection(int aStartSpan, int
   aStartChannel, int anEndSpan, int anEndChannel, int
   aConID);
int sk_unMonitorChannelOnConnection(int aStartSpan, int
   aStartChannel, int anEndSpan, int anEndChannel, int
   aConID);
```

Threadsafe Syntax

```
int skts_monitorChannel( int aStartSpan, int
   aStartChannel, int anEndSpan, int anEndChannel, int
   aConID );
int skts_unMonitorChannel( int aStartSpan, int
   aStartChannel, int anEndSpan, int anEndChannel, int
   aConID );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aStartSpan	Specifies the first span to be monitored.
aStartChannel	Specifies the first channel to be monitored.
anEndSpan	Specifies the last span to be monitored.
anEndChannel	Specifies the last channel to be monitored.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_removeChannelsFromGroup()

Description

The ***sk_removeChannelsFromGroup***() function is used to remove physical channels from channel groups. This function is also available as ***sk_removeChannelsFromGroupOnConnection()*** and ***skts_removeChannelsFromGroup()***.

Syntax

```
int sk_removeChannelsFromGroup(char *aChannelGroup, int
   aStartSpan, int aStartChannel, int anEndSpan, int
   anEndChannel);
int sk_removeChannelsFromGroupOnConnection(char
   *aChannelGroup, int aStartSpan, int aStartChannel, int
   anEndSpan, int anEndChannel, int aConID);
int skts_removeChannelsFromGroup( char *aChannelGroup,
   int aStartSpan, int aStartChannel, int anEndSpan, int
   anEndChannel, int aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Identifies the channel group.
aStartSpan	Identifies the first span in the group.
aStartChannel	Identifies the first chan in the group.
anEndSpan	Identifies the last span in the group.
anEndChannel	Identifies the last chan in the group.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_requestChannel()

Description To find an unused channel in the group, your application must call ***sk_requestChannel()***. If successful, the application receives notification of the channel it was assigned. The channel can receive inbound calls, the application must be prepared to handle an inseizure before the channel can be used. Inseizure is indicated by the receipt of a *RequestForService* or *RFSWithData* message.

This function causes ***SK_RequestChannelAck***, an acknowledgment message, to be sent to your application after processing has been completed. This message contains the span and channel allocated to the application module. This channel is now allocated to the application module, and all future messages for that channel are routed to this process.

The aT*ag* and aHandlerFunc are equivalent to the tag and handler function arguments passed to ***sk_sendMsgStruct()***.

> **Important!** To request a channel using the threadsafe library, use ***skts_requestOutseizedChannel()***.

This function is also available as ***sk_requestChannelOnConnection()***.

Syntax

```
int sk_requestChannel(char *aChannelGroup, void *aTag,
   HandlerFunc *aHandlerFunc);
int sk_requestChannelOnConnection(char *aChannelGroup,
   void *aTag, HandlerFunc *aHandlerFunc, int aConID);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aChannelGroup	*group specifies the channel group.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_NO_CHANNELS	There are no channels available.

sk_requestOutseizedChannel()

Description

This function is similar to ***sk_requestChannel()***, except that it automatically outseizes the allocated channel with the ***XL_OutseizeControl*** message before returning it. If successful, the application receives notification of the channel it was assigned, in addition to any outseize acknowledgments. If there is glare or some other error condition, the LLC automatically tries another channel. This process is repeated up to nRetries times. If there is a problem with all the channels tried, the application is notified with a message containing the last error condition. The *aTag* and *aHandlerFunc* are comparable to the tag and handler function arguments passed to ***sk_sendMsgStruct***.

This function causes a ***RequestChannelAck*,** an acknowledgment message, to be sent to your application after processing has been completed. This message contains the span and channel allocated to the application module. The channel is now allocated to the application module, and all future messages for that channel are routed to this process.

In order to request a channel without performing an outseize, enter a NULL as the *anOutseizeMsg* argument.

This function is also available as ***sk_requestOutseizedChannelOnConnection()*** and ***skts_requestOutseizedChannel***.

Syntax

```
int sk_requestOutseizedChannel (char *aChannelGroup, int
   aNumRetries, XL_OutseizeControl *anOutseizeMsg, void
   *aTag, HandlerFunc *aHandlerFunc);
int sk_requestOutseizedChannelOnConnection(char
   *aChannelGroup, int aNumRetries, XL_OutseizeControl
   *anOutseizeMsg, void *aTag, HandlerFunc *aHandlerFunc,
   int aConID);
int skts_requestOutseizedChannel( char *aChannelGroup,
   int aNumRetries, XL_OutseizeControl *anOutseizeMsg,
   void *aTag, HandlerFunc *aHandlerFunc, int aConID);
```

Parameters

The function parameters are shown in the next table.

Argument	Description
aChannelGroup	Specifies the channel group
aNumRetries	Limits the number of attempts to find a channel before an error gets send to the application module.
anOutseizeMsg	Pointer to switch API message
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC — This return value indicates your application lost contact with the LLC.

SK_INVALID_OUTSEIZE — This return value indicates the channel group is not configured for outseize traffic.

SK_CHANNEL_PURGING — This return value indicates the channel purged while being outseized by the LLC.

sk_requestRouteControlledChannel()

Description

This function is similar to ***sk_requestChannel*( )**, except that it automatically outseizes the allocated channel with the ***XL_RouteControl*** message before returning it. If successful, the application receives notification of the channel it was assigned, in addition to any acknowledgments to the route control request. If there is glare or some other error condition, the LLC automatically tries another channel. This process is repeated up to nRetries times. If there is a problem with all the channels tried, the application is notified with a message containing the last error condition. The aT*ag* and *aHandlerFunc* are comparable to the tag and handler function arguments passed to ***sk_sendMsgStruct***.

This function causes ***SK_RouteControlledChannelAck*,** an acknowledgment message, to be sent to your application after processing has been completed. This message contains the span and channel allocated to the application module. The channel is now allocated to the application module, and all future messages for that channel are routed to this process.

This function is also available as ***sk_requestRouteControlledChannelOnConnection()*** and ***skts_requestRouteControlledChannel***.

Syntax

```
int sk_requestRouteControlledChannel(char *aChannelGroup,
   int aNumRetries, XL_RouteControl *aRouteControlMsg,
   void *aTag, HandlerFunc *aHandlerFunc);
int sk_requestRouteControlledChannelOnConnection(char
   *aChannelGroup, int aNumRetries, XL_RouteControl
   *aRouteControlMsg, void *aTag, HandlerFunc
   *aHandlerFunc, int aConID);

int skts_requestRouteControlledChannel( char
   *aChannelGroup, int aNumRetries, XL_RouteControl
   *aRouteControlMsg, void *aTag, HandlerFunc
   *aHandlerFunc, int aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Specifies the channel group.
aNumRetries	Limits the number of attempts to find a channel before an error gets send to the application module.

Argument	Description
aRouteControlMsg	Pointer to EXS API message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_returnChannel()

Description

Your application should call ***sk_returnChannel***() when a process is finished with a channel. This function returns the channel to the pool of free channels, and stops the process from receiving messages for that channel.

To prevent future messages from coming to the application module unexpectedly, the function ***sk_returnChannel***() must be called for channels explicitly assigned from ***sk_requestOutseizedChannel***() and implicitly assigned through ***sk_watchChannelGroup***().

Alternatively, if the environment variable, SK_AUTO_RETURN_CHANNELS is set, any time the message ***XL_ChannelReleased*** is sent from the switch, the channel is returned automatically. We recommend using SK_AUTO_RETURN_CHANNELS because it simplifies application development. It also prevents a possible race condition when a ***XL_ChannelReleased*** message is immediately followed by a ***XL_RequestForService***, before the application has been able to return the channel. To prevent an application from returning channels allocated to other applications, set the environment variable, SK_RETURN_CHANNEL_BY_OWNER_ONLY.

This function is also available as ***sk_returnChannelOnConnection()*** and ***skts_returnChannelOnConnection()***.

Syntax

```
int sk_returnChannel(int aSpan, int aChannel);
int sk_returnChannelOnConnection(int aSpan, int aChannel,
   int aConID);
int skts_returnChannel(int aSpan, int aChannel, int
   aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aSpan	Identifies the span number.
aChannel	Identifies the channel number.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_NOT_ ALLOCATED	This return value indicates that the channel is not allocated to this application.

sk_setChannelHandler()

Description You can use the ***sk_setChannelHandler***() function to organize the flow of messages for your application. Once a channel has been assigned to an application, the ***sk_setChannelHandler***() function causes the specified handler function to be called whenever a CSP-initiated message arrives on a span and channel. This function does not affect the routing of any acknowledgment messages.

Syntax

```
int sk_setChannelHandler(int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHanderFunc);
int skts_setChannelHandler( int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHanderFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	Span identifies the span number.
aChannel	Chan identifies the channel number.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Always returns OK.

sk_setChannelData() / sk_getChannelData()

Description

This function ***sk_setChannelData***() allows you to associate arbitrary data with a specific channel. To clear the data associated with a channel, just enter `NULL` as data.

The function ***sk_getChannelData***() allows the retrieval of arbitrary data that was stored with ***sk_setChannelData***(). If nothing was ever set, then the function returns `NULL`.

Syntax

```
int sk_setChannelData(int aSpan, int aChannel, void
   *someData);
int sk_getChannelData(int aSpan, int aChannel);
```

Threadsafe Syntax

```
void skts_setChannelData( int aSpan, int aChannel, void
   *someData);
void skts_getChannelData( int aSpan, int aChannel);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aSpan	Indicates the span number.
aChannel	Indicates the channel number.
someData	Data pointer.

sk_setGroupHandler()

Description Use the ***sk_setGroupHandler***() function to complement the function ***sk_watchChannelGroup***(). When a message arrives on a channel that does not have an explicitly-assigned handler function, the handler function of its group is notified.

Syntax

```
int sk_setGroupHandler(char *aChannelGroupName, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_setGroupHandler(char *aChannelGroupName, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aChannelGroupName	The name of a specified channel group.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

How sk_setGroupHandler works Send the following function calls, with respect to your naming convention, to the LLC:

sk_watchChannelGroup(“inbound”);
sk_setGroupHandler(“inbound”, NULL, handleInboundCall);
After this function call is sent and if your application uses ***sk_rcvAndDispatch***() or ***sk_rcvAndDispatchAutoStorage***(), every time a message is received for a channel in the specified group, the message will be sent to this handler function.

Return Values Always returns OK.

sk_watchChannelGroup()

Description

Use the function ***sk_watchChannelGroup()*** if your application handles messages that come in on any of the channels associated with the specified channel group. The LLC then routes all messages that come in on channels belonging to the specified group to your application module. The function is typically used on inbound channels.

If your application is watching a channel group, it is responsible for responding to events on those channels.

In case of a disconnect from the LLC, the LLC logs information about the channel groups the application was watching.

This function is also available as ***sk_watchChannelGroupOnConnection()*** and ***skts_watchChannelGroup***.

Syntax

```
int sk_watchChannelGroup(char *aChannelGroup);
int sk_watchChannelGroupOnConnection(char *aChannelGroup,
   int aConID);
int skts_watchChannelGroup( char *aChannelGroup, int
   aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aChannelGroup	Points to the group of channels the application will watch.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC — This return value indicates that your application lost contact with the LLC.

API messages used for Channel Management

API Messages

The API messages listed below are used for channel management.The API messages are found in the *API Reference*.

- AllocateChannel
- AllocateChannelGroup
- BroadcastLoad
- ChannelGroupContentsQuery
- ChannelGroupPopulationQuery
- ClearChanGroup
- ClearAllChanGroups
- ConfigChanGroup
- IgnoreChanGroup
- Remove Channels From Group
- TransferChanMsg
- WatchChanGroup

4 Register Functions

Purpose

This chapter describes the register functions available in SwitchKit. It provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

The register functions allow your application to monitor different areas of the communication in your system. The registration lets the LLC route messages, specified through the function call, to your application, so that your application can handle them.

sk_appGroupRegister()

Description

The ***sk_appGroupRegister***() function allows a process register to receive messages directed to the specified application group. This application group is a text name. The application group name can be used in both ***InterAppMsg*** and ***TransferChanMsg***. Whenever a message is sent to an application group, the LLC routes that message to the least loaded application (as specified through ***sk_broadcastLoad***()) that has registered for that application group. This function can be used to allow client applications to refer to servers by a text name instead of by a number. Furthermore, there can be multiple servers set up to load share application requests.

For example, you could build a database server that receives SwitchKit messages, does database queries, and replies. Instead of defining a single application number of the database server, you can instead use the string “databaseServer” in all ***InterAppMsg*** database requests. Whenever you run a database server, it registers for that application group and is available to receive database requests. If it is feasible to run multiple database servers, the requests are load shared among the available servers.

This message is also available as ***sk_appGroupRegisterOnConnection()*** and ***skts_appGroupRegister()***.

Syntax

```
int sk_appGroupRegister(const char *virtualName);

int sk_appGroupRegisterOnConnection(const char
   *virtualName, int conID);
```

Threadsafe Syntax

```
int skts_appGroupRegister( const char * aVirtualName, int
   aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aVirturalName	A string identifier for an application group which can be targets for an *InterAppMsg* or *TransChanMsg*.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC — This return value indicates that your application lost contact with the LLC.

Useful Related Functions

SK_Connection ****sk_createConnectionWithID()***;

sk_msgRegister() / sk_msgUnRegister()

Description

The ***sk_msgRegister***() function allows your application to receive switch or LLC initiated messages that are not associated with a channel. Registering for these messages does not affect the flow of messages to SwitchManager or to any other process. This message is also available as ***sk_msgRegisterOnConnection()*** and ***skts_msgRegister()***.

The ***sk_msgUnRegister***() function cancels the ***sk_msgRegister***() call. This message is also available as ***sk_msgUnRegisterOnConnection()*** and ***skts_msgUnRegister()***.

Syntax

```
int sk_msgRegister(int aMsgRegisterMask);
int sk_msgUnRegister(int aMsgRegisterMask);
int sk_msgRegisterOnConnection(int aMsgRegisterMask, int
   aConID);
int sk_msgUnRegisterOnConnection(int aMsgRegisterMask,
   int aConID);
```

Threadsafe Syntax

```
int skts_msgRegister( int aMsgRegisterMask, int aConID );
int skts_msgUnRegister( int aMsgRegisterMask, int
   aConID);
```

Parameters

The function parameters are shown in the table below.

Important! All of the bit mask values are defined in SK_API.h. All of the message structures are defined in Messages.api.h. The C++ Classes are defined in CMessages.api.h. All of the messages are documented in the *API Reference*.

Argument - Bit Mask Value defined in SK_API.h	Message	Originator
SK_ALL_MSGS	All messages	Switch/LLC
SK_POLLS	PollMessage	Switch
SK_ALARMS	Alarm, AlarmCleared	Switch
SK_CSRS	CardStatusReport	Switch
SK_NSRS	NodeStatusReport	Switch
SK_RSRS	RingStatusReport	Switch
SK_DS0S	DS0StatusChange	Switch

Argument - Bit Mask Value defined in SK_API.h	Message	Originator
SK_PPLEIS	PPLEventIndication	Switch
SK_CONFERENCE_DELETED	ConferenceDeleted ConferenceDeleteRequest	Switch Host
SK_CONFIGMSGS (Internal Use Only)	All configuration messages	Client
SK_CONNECT_STATUS	ConnectionStatusMsg	LLC
SK_CHAN_PROBLEM	ChannelProblem	LLC
SK_CONFIG_STATUS	ConfigStatusMsg	SwitchManager
SK_CPE_ON_CONFERENCE	CallProcessingEvent messages which contain a conference AIB.	Switch
SK_RESOURCE_UTIL_ REPORT	GenericReport (Resource utilization report)	LLC
SK_D_SERVER_REPORT	DeviceServerStatus Report	Switch
SK_HOSTALARM	HostAlarm	LLC
SK_SERVER_STATUS_ REPORT	ServerStatusReport	Switch
SK_SERVER_HOST_POLL	ServerHostPoll	Switch
SK_SERVER_STATUS_ CHANGE	ServerStatusChange	LLC
SK_ARP_CACHE_REPORT	ARPCacheReport	Switch
SK_CCS_REP	CCSRedundancy Report	Switch

Argument	Description
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_pplComponentRegister() / sk_pplComponentUnRegister()

Description

Use the ***sk_pplComponentRegister***() function to receive ***PPLEventIndication*** messages for a particular component. This message is also available as ***sk_pplComponentRegisterOnConnection()*** and ***skts_pplComponentRegister()***.

Use the ***sk_pplComponentUnRegister***() function to unregister to receive ***PPLEventIndication*** messages for a particular component. This message is also available as ***sk_pplComponentUnRegisterOnConnection()*** and ***skts_pplComponentUnRegister()***.

Syntax

```
int sk_pplComponentRegister(int aComponentID);
int sk_pplComponentUnRegister(int aComponentID);
int sk_pplComponentRegisterOnConnection(int aComponentID,
   int aConID);
int sk_pplComponentUnRegisterOnConnection(int
   aComponentID, int aConID);
```

Threadsafe Syntax

```
int skts_pplComponentRegister( int aComponentID, int
   aConID );
int skts_pplComponentUnRegister( int aComponentID, int
   aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aComponentID	Specifies the PPL component number.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

The possible return values for this function:

SK_LOST_LLC — This return value indicates that your application lost contact with the LLC.

sk_pplTCAPRegister() / sk_pplTCAPUnRegister()

Description

Use the function ***sk_pplTCAPRegister***() to enable an application to register for TCAP traffic by Sub-System Number (SSN) or Origination Point Code(OPC). This message is also available as ***sk_pplTCAPRegisterOnConnection()*** and ***skts_pplTCAPRegister()***.

Use ***sk_pplTCAPUnRegister()*** to unregister. This message is also available as ***sk_pplTCAPUnRegisterOnConnection()*** and ***skts_pplTCAPUnRegister()***.

Syntax

```
int sk_pplTCAPRegister(int aRegisterVal, UBYTE
   aRegisterType);
int sk_pplTCAPUnRegister(int aRegisterVal, UBYTE
   aRegisterType);
int sk_pplTCAPRegisterOnConnection(int aRegisterVal,
   UBYTE aRegisterType, int aConID);
int sk_pplTCAPUnRegisterOnConnection(int aRegisterVal,
   UBYTE aRegisterType, int aConID)
```

Threadsafe Syntax

```
int skts_pplTCAPRegister( int aRegisterVal, UBYTE
   aRegisterType, int aConID );
int skts_pplTCAPUnRegister( int aRegisterVal, UBYTE
   aRegisterType, int aConID );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aRegisterVal	OPC or Sub-System Number
aRegisterType	Registration type. Use the following constants to specify either OPC or SSN: SK_TCAP_REG_OPC (0x00) - OPC Registration SK_TCAP_REG_SSN (0x01) - SSN Registration SK_TCAP_REG_STACK_SSN 0x02 - Registers a combination of stack and subsystem number.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

5 Logging Functions

Purpose

This chapter describes the logging functions available in SwitchKit. It provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

All modules within SwitchKit generate log files. You can also generate your own log files, and have them managed by SwitchKit. Log file entries can also be displayed on the screen. By default, SwitchManager and the LLC send their output to the screen as well as to the log file, while the applications only write it to the log file. The behavior of the LLC and SwitchManager can be changed through the environment variable SK_SCREEN_OUTPUT, while the behavior of the applications can be changed by using the function ***sk_setSilentMode()***.

sk_getMsgName()

Purpose Use the ***sk_getMsgName()*** function to return the name of the given message to your application. This function is also available as ***skts_getMsgName()***.

Description This function call returns the name of a message to the application.

Syntax

```
int sk_getMsgName(MsgStruct *);
```

Threadsafe Syntax

```
int skts_getMsgName(MsgStruct *);
```

Parameters The function parameters are shown in the table below.

Argument	Description
MsgStruct *	Pointer to a generic C Structure representing a message.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
“Message”	If the message is a known Toolkit message or EXS API message, the function returns the message name.
Unknown TK Tag	The message is an unknown Toolkit message, returned with the message tag.
Unknown Tag	The message is an unknown EXS API message, returned with the message tag.
AdminMessage	The message is an AdminMessage.
DummyMessage	If the message is a DummyMessage.

sk_logLevel()

Purpose Use the ***sk_logLevel***() function to manipulate the level of logging in your system. This function is also available as ***sk_logLevelOnConnection() and skts_logLevel***.

Description This function changes the level of logging based on different settings.

Syntax

```
int sk_logLevel(int aLogLevel);
int sk_logLevelOnConnection(int aLogLevel, int aConID);
```

Threadsafe Syntax

```
int skts_logLevel(int aLogLevel, int aConID);
```

Parameters The function parameters are shown in the table below.

CAUTION

Pay close attention to Bit 4 behavior. By default the screen output is enabled, although the bit is cleared. The screen output is disabled only if you set the bit.

Argument	Description
aLogLevel	what is a bit mask represented by the following bits: • Bit 1 is set - socket.log is turned on. • Bit 2 is set - messages.log is turned on. • Both bits cleared - logging is turned off. • Bit 4 is set - screen output is turned off.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with..

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_logMessage()

Purpose Use the ***sk_logMessage***() function to log a specific message to a specified log level. This function is also available as ***sk_logMessageOnConnection()*** and ***skts_logMessage***.

Description This function logs a specified message to a specified log level. This function writes to the Switchmgr Alarm.log.

Syntax

```
int sk_logMessage(char *aTextMessage, int aLogLevel);
int sk_logMessageOnConnection(char *aTextMessage,int
   aLogLevel, int aConID);
```

Threadsafe Syntax

```
int skts_logMessage(char *aTextMessage,int aLogLevel, int
   aConID);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTextMessage	Points to the message you want to log.
aLogLevel	This can be specified as follows: SK_LVL_INFORM = 0 SK_LVL_MAJOR = 1 SK_LVL_MINOR = 2 SK_LVL_INFORM - simply log the message, no error reported SK_LVL_MAJOR - log the message as a major error SK_LVL_MINOR - log the message as a minor error
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with..

Return Values Possible return values for this function:

OK	Successfully executed.

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_setSilentMode()

Purpose Use the ***sk_setSilentMode()*** function to cause or prevent log output from appearing on the screen. This function is also available as ***sk_setSilentMode()***.

Description This function controlls the log out put on the screen. By default, log output from the application appears only in the log files.

Syntax

```
int sk_setSilentMode(int isSilentModeFlag);
```

Threadsafe Syntax

```
void skts_setSilentMode(int isSilentModeFlag);
```

Parameters The function parameters are shown in the table below.

Argument	Description
isSilentModeFlag	A non-zero value disables the output of logs to the screen. Setting it to 0 causes all output to log files to also appear on the screen.

Return Values This function does not have a return value.

6 Handler Functions

Purpose This chapter describes the handler functions available in SwitchKit. It provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

Handler Functions

Description

To assist with the standard state machines, the SwitchKit API allows for incoming messages to be automatically dispatched to an application that has assigned handler functions. Using these functions, you can install functions to handle state transitions. In this way, you can avoid using multiple threads to manage different ports, because SwitchKit handles that automatically.

> **Important!** When you write you handler functions, make sure that you register for the right ***sk_rcvAndDispatch***() function.

Definition of Handler Functions

```
typedef struct {
   MsgStruct *IncomingMsg;
   MsgStruct *AckedMsg;
   void *UserData;
   } SK_Event;
```

Syntax

```
int HandlerFunc(SK_Event *evt, void *tag);
```

How Applications use Handler Functions

There are four ways that an application can specify the handler function that is dispatched. Three of them deal with switch-initiated messages, and the fourth deals with acknowledgments to host-initiated messages.

By using the following handler functions, SwitchKit can take care of the bookkeeping associated with a state machine. Context associated with a call is passed from one state to the next through the tag, and all acknowledgments are augmented by a pointer to the original message.

Switch-Initiated

- sk_setChannelHandler();
- sk_pushChannelHandler();
- sk_setGroupHandler();

Acknowledgment to Host-Initiated

- sk_sendMessageWithHandler();

Default Handler Functions

The default handler functions can be used to set up default handlers, to be called if no other handler is appropriate. For example, if ***sk_rcvAndDispatch***() returned SK_NOT_HANDLED, you can add a default handler. This function can be used to set up handlers for Inter

Application messages, switch-initiated messages such as polls and alarms, or as acknowledgments to any message that was sent without an explicitly-specified handler for its acknowledgment.

Default Handlers

- sk_setDefaultHandler();
- sk_pushDefaultHandler();
- sk_popDefaultHandler();
- sk_removeDefaultHandler();

Important! If the SK_HANDLER_BEHAVIOR bit mask has the SK_HDLR_BHVR_SAFETY_NET bit set, the default handler can also be invoked if the appropriate handler is found but returns SK_NOT_HANDLED.

Handler Function Stacks

For simple applications, it is usually sufficient to have just a single handler function associated with any channel. For more complicated applications, more power is often needed. SwitchKit allows a stack of handlers to be associated with a channel. These facilities are demonstrated in the *callcard.c* application included in the distribution. Multiple handler functions can be added onto a channel's handler function stack. When a switch-initiated message arrives on that channel, the most recently added handler is called first. That handler can choose to not handle the message by returning SK_NOT_HANDLED, in which case the next handler on the stack is called.

The *SK_Event* that is passed to the handler function contains a UserData field. This field can be used to pass information within the stack of handler functions that are called for a single event. It is initialized to NULL.

By using a stack of handlers, a series of specialized handlers can be added that only handle a small set of messages. The first handler that is added should be a generic handler, to handle all messages that haven't been handled at a lower level. Lower level handlers can set some state information in the UserData field of the `SK_Event` in order to communicate that part of an otherwise unhandled message has been processed.

Example

The ***sk_pushChannelHandler***()function pushes the handler function, `aHandlerFunc`, onto the handler stack for the specified span and channel.

The ***sk_popChannelHandler***() function removes the top of the Handler Stack for this span, channel and sets `aHandlerFunc` to point to this function.

The ***sk_removeChannelHandler***() function removes the specific Handler Function, `aHandlerFunc` from the stack for this span, channel.

```
sk_pushHandler(span,chan,genericHandler);
sk_pushHandler(span,chan,h);
int genericHandler(SK_Event *e, void *tag)
   {
      MsgStruct *m = e->IncomingMsg;
      bool beenPrinted = (bool)e->UserData;
         if (!beenPrinted) {
            e->UserData = (void *)1;
            printMessage(m);
         }
      printf (“Unhandled message %s\n”,sk_getMsgName(m));
}
int h(SK_Event *e, void *tag)
   {
      MsgStruct *m = e->IncomingMsg;
      bool beenPrinted = (bool)e->UserData;
         if (!beenPrinted) {
            e->UserData = (void *)1;
            printMessage(m);
         }
      SK_MSG_SWITCH(m)
      {
         CASE_RequestForService(rfs)
         {
            handleRFS(rfs);
            return OK;
         }
      } SK_END_SWITCH;
   return SK_NOT_HANDLED;
}
```

sk_popChannelHandler()

Description The ***sk_popChannelHandler***() function removes the top of the handler stack for the specified span and channel and sets `aHandlerFunc` to point to this function.

Syntax

```
int sk_popChannelHandler(int aSpan, int aChannel, void
   **aRtndTag, HandlerFunc **aRtndHanderFunc);
```

Threadsafe Syntax

```
int skts_popChannelHandler(int aSpan, int aChannel, void
   **aRtndTag, HandlerFunc **aRtndHanderFunc
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	Specifies the span.
aChannel	Specifies the channel.
aRtndTag	Output parameter that will contain the tag value that was originally specified when the handler was pushed on the handler stack.
aRtndHandlerFunc	Output parameter that will contain the message handler function that was originally specified when the handler was pushed on the handler stack.

Return Values Possible return values for this function:

SK_EMPTY	The handler stack is empty.

sk_pushChannelHandler()

Description The ***sk_pushChannelHandler()*** function pushes the handler function `aHandlerFunc` onto the handler stack for the specified span and channel. This function is also available as ***skts_pushChannelHandler()***.

Syntax

```
int sk_pushChannelHandler(int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHanderFunc);
```

Threadsafe Syntax

```
int skts_pushChannelHandler( int aSpan, int aChannel,
   void *aTag, HandlerFunc* aHanderFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	Secifies the span.
aChannel	Specifies the channel.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Always returns OK.

sk_removeChannelHandler()

Description The ***sk_removeChannelHandler***() function removes the handler function `aHandlerFunc` from the stack for the specified span and channel. This function is also available as ***skts_removeChannelHandler()***.

Syntax

```
int sk_removeChannelHandler(int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_removeChannelHandler(int aSpan, int aChannel,
   void *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	sp specifies the span.
aChannel	ch specifies the channel.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_NOT_PRESENT — The specified handler function is not in the handler stack of the channel.

sk_setChannelHandler()

Description The ***sk_setChannelHandler***() function installs `aHandlerFunc` as the handler. If there is currently a stack of handler functions, it is cleared first. This function is also available as ***skts_setChannelHandler()***.

Syntax

```
int sk_setChannelHandler(int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_setChannelHandler(int aSpan, int aChannel, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aSpan	Specifies the span.
aChannel	Specifies the channel.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Always returns OK.

sk_setDefaultHandler()

Description The ***sk_setDefaultHandler***() function installs `aHandlerFunc` as the default handler. If there is currently a stack of handler functions, it is cleared first. This function is also available as ***skts_setDefaultHandler()***.

Syntax

```
int sk_setDefaultHandler(void *aTag, HandlerFunc*
   aHandlerFunc );
```

Threadsafe Syntax

```
int skts_setDefaultHandler(void *aTag, HandlerFunc*
   aHandlerFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_setGroupHandler()

Description The ***sk_setGroupHandler***() function installs HandlerFunc as the group handler. If there is currently a stack of handler functions, it is cleared first. This function is also available as ***skts_setGroupHandler()***.

Syntax

```
int sk_setGroupHandler(char *aChannelGroupName, void
   *aTag, aHandlerFunc*);
```

Threadsafe Syntax

```
int skts_setGroupHandler( char *aChannelGroupName, void
   *aTag, HandlerFunc* aHandlerFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aGroupName	Specifies the group.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

sk_pushDefaultHandler()

Description The ***sk_pushDefaultHandler***() function pushes `aHandlerFunc` onto the stack of default handler functions. It is called before any existing handler functions on the stack. If no other handler functions apply, then the next highest function on the stack is called. This function is also available as ***skts_pushDefaultHandler()***.

Syntax

```
int sk_pushDefaultHandler(void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_pushDefaultHandler( void *aTag, HandlerFunc*
   aHandlerFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_popDefaultHandler()

Description The ***sk_popDefaultHandler***() function removes the top-most handler function from the default handler function stack. This function is also available as ***skts_popDefaultHandler()***.

Syntax

```
int sk_popDefaultHandler(void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_popDefaultHandler( void *aTag, HandlerFunc*
   aHandlerFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_removeDefaultHandler()

Description The ***sk_removeDefaultHandler***() function removes the specified handler `aHandlerFunc` from the handler function stack wherever it is. This function is also available as ***skts_removeDefaultHandler()***.

Syntax

```
int sk_removeDefaultHandler(void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_removeDefaultHandler( void *aTag, HandlerFunc*
   aHandlerFunc );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_setLLCConnectionHandler()

Description

The ***sk_setLLCConnectionHandler***() function installs a handler function that will be called any time the application's connection with the LLC is created or destroyed by the SwitchKit API. The connection is created each time the application successfully connects to the LLC. The connection is destroyed each time the application is disconnected from the LLC. Disconnection may occur for any number of reasons including, termination of the LLC, network problems, or failure of the application to call rcvAndDispatch() on a regular basis.

When an LLC switchover occurs, the LLC connection handler will be called twice:

- When the connection to the LLC is initially lost indicating that a connection has been destroyed
  (state = SK_LLC_CONN_DESTROYED)
- When the connection to an LLC is established indicating that a connection has been created
  (state = SK_LLC_CONN_CREATED).

This function is also available as ***skts_removeLLCConnectionHandler()***.

Syntax

```
int sk_setLLCConnectionHandler( void *aTag,
   LLCHandlerFunc *anLLCHandlerFunc);
```

Threadsafe Syntax

```
int skts_setLLCConnectionHandler( void *aTag,
        LLCHandlerFunc *anLLCHandlerFunc);
```

Definition of LLC Connection Handler Functions

Syntax

```
void LLCHandlerFunc(int aConId, int aConState, void
   *aTag);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aConId	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Argument	Description
aConState	The new state of the LLC connection. Possible values are: SK_LLC_CONN_CREATED Application connection to LLC has just been established. SK_LLC_CONN_DESTROYED Application connection to LLC has just been lost.
aTag	A value to be passed into the LLC connection handler each time the LLC connection handler is called. This value is also used to reference the handler when attempting to remove a handler using sk_removeLLCConnectionHandler().

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	A value to be passed into the LLC connection handler each time the LLC connection handler is called. This value is also used to reference the handler when attempting to remove a handler using sk_removeLLCConnectionHandler().
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_HANDLER_ALREADY_EXISTS	A connection handler for this tag already exists. Please remove the old connection handler before setting new handler.
SK_UNABLE_TO_OBTAIN_LOCK	Error occurred obtaining a Mutex (Threadsafe library only.) OK - Handler successfully set.
OK	Success

sk_removeLLCConnectionHandler()

Description The ***sk_removeLLCConnectionHandler***() function removes the LLC connection handler associated with the specified tag value. This function is also available as ***skts_removeLLCConnectionHandler()***.

Syntax

```
int sk_removeLLCConnectionHandler( void *aTag,
   LLCHandlerFunc *anLLCHandlerFunc);
```

Threadsafe Syntax

```
int skts_removeLLCConnectionHandler( void *aTag,
   LLCHandlerFunc *anLLCHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	The value used when the LLC connection handler was originally defined using sk_setLLCConnectionHandler(). If the same value is not used, the LLC connection handler will not be removed.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

SK_HANDLER_NOT_FOUND	A connection handler with the specified tag was not found. No connection handler was removed.
SK_UNABLE_TO_OBTAIN_LOCK	Error occurred obtaining a Mutex. (Threadsafe library only.) OK - A handler was successfully removed.
OK	Success

sk_popDefaultTCAPHandler()

Description Use the ***sk_popDefaultTCAPHandler***() function to remove the top-most handler function from the default TCAP handler function stack. This function is also available as ***skts_popDefaultTCAPHandler***().

Syntax

```
int sk_popDefaultTCAPHandler( void **aRtndTag,
   HandlerFunc **aRtndHandlerFunc);
```

Threadsafe Syntax

```
int skts_popDefaultTCAPHandler( void **aRtndTag,
   HandlerFunc **aRtndHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	Output parameter that will contain the tag value that was originally specified when the handler was pushed on the handler stack.
aRtndHandlerFunc	Output parameter that will contain the message handler function that was originally specified when the handler was pushed on the handler stack.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully popped.
SK_EMPTY	Unable to pop handler.

sk_popTCAPHandler()

Description Use the ***sk_popTCAPHandler***() function to remove the top of the TCAP handler stack for the specified stack and subsystem number and set aRtndHandlerFunc to point to this function. This function is also available as ***skts_popTCAPHandler***().

Syntax

```
int sk_popTCAPHandler(int aStackID, int anSSN, void
   **aRtndTag, HandlerFunc **aRtndHandlerFunc);
```

Threadsafe Syntax

```
int skts_popTCAPHandler(int aStackID, int anSSN, void
   **aRtndTag, HandlerFunc **aRtndHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aStackID	Stack ID
anSSN	Sub-system number
aRtndTag	Output parameter that will contain the tag value that was originally specified when the handler was pushed on the handler stack.
aRtndHandlerFunc	Output parameter that will contain the message handler function that was originally specified when the handler was pushed on the handler stack.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully popped.
SK_EMPTY	Unable to pop handler.

sk_pushDefaultTCAPHandler()

Description Use the ***sk_pushDefaultTCAPHandler***() function to push aHandlerFunc onto the stack of default TCAP handler functions. It is called before any existing handler functions on the stack. If no other handler functions apply, then the next highest function on the stack is called. This function is also available as ***skts_pushDefaultTCAPHandler()***.

Syntax

```
int sk_pushDefaultTCAPHandler( void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_pushDefaultTCAPHandler( void *aTag, HandlerFunc*
   aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully pushed.
SK_BAD_LICENSE	TCAP handler functionality was not successfully unlocked.

sk_pushTCAPHandler()

Description Use the ***sk_pushTCAPHandler***() function to push the handler function, `aHandlerFunc`, onto the TCAP handler stack for the specified stack and subsystem number. This function is also available as ***skts_pushTCAPHandler()***.

Syntax

```
int sk_pushTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_pushTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aStackID	Stack ID
anSSN	Sub-system number
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully pushed.
SK_BAD_LICENSE	TCAP handler functionality was not successfully unlocked.

sk_removeDefaultTCAPHandler()

Description Use the ***sk_removeDefaultTCAPHandler***() function to remove the specified handler, `aHandlerFunc`, from the Default TCAP handler function stack wherever it is. This function is also available as ***skts_removeDefaultTCAPHandler()***.

Syntax

```
int sk_removeDefaultTCAPHandler(void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_removeDefaultTCAPHandler(void *aTag,
   HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

TRUE	Always returns TRUE (1)

sk_removeTCAPHandler()

Description Use the ***sk_removeTCAPHandler***() function to remove the handler function, `aHandlerFunc`, from the TCAP handler stack for the specified stack and subsystem number. This function is also available as ***skts_removeTCAPHandler()***.

Syntax

```
int sk_removeTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_removeTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aStackID	Stack ID
anSSN	Subsystem number
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

TRUE	Always returns TRUE (1)

sk_setDefaultTCAPHandler()

Description Use the ***sk_setDefaultTCAPHandler***() function to install `aHandlerFunc` as the default TCAP handler. If there is currently a stack of default TCAP handler functions, it is cleared first.. This function is also available as ***skts_setDefaultTCAPHandler()***.

Syntax

```
int sk_setDefaultTCAPHandler( void *aTag, HandlerFunc*
   aHandlerFunc);
```

Threadsafe Syntax

```
int skts_setDefaultTCAPHandler( void *aTag, HandlerFunc*
   aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully set.
SK_BAD_LICENSE	TCAP handler functionality was not successfully unlocked.

sk_setTCAPHandler()

Description Use the ***sk_setTCAPHandler***() function to organize the flow of TUSI and SUSI messages for your application. Once a stack and subsystem number have been assigned to an application, the sk_setTCAPHandler() function causes the specified handler function to be called whenever a CSP-initiated message arrives on a stack and subsystem number. This function does not affect the routing of any acknowledgment messages. This function is also available as ***skts_setTCAPHandler()***.

Syntax

```
int sk_setTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Threadsafe Syntax

```
int skts_setTCAPHandler(int aStackID, int anSSN, void
   *aTag, HandlerFunc* aHandlerFunc);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aStackID	Identifies Stack ID
anSSN	Identified Subsystem number
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Return Values Possible return values for this function:

OK	Always returns OK.

sk_unlockTCAPHandlers()

Description Use the ***sk_unlockTCAPHandlers()*** function to specify an Dialogic-defined value that will make the TCAP handlers accessible to Switchkit applications..This function is also available as ***skts_unlockTCAPHandlers().***

Syntax

```
sk_unlockTCAPHandlers(char *aKey);
```

Threadsafe Syntax

```
skts_unlockTCAPHandlers(char *aKey)
```

Parameters The function parameters are shown in the table below.

Argument	Description
aKey	Dialogic-defined value used to unlock TCAP handlers.

Return Values Possible return values for this function:

OK	TCAP handler functionality was successfully unlocked.
SK_NOT_PRESENT	This return value indicates an invalid key.

7 Message Functions

Purpose This chapter provides information about message structures and macros, as well as information on sending and receiving message functions that are available in SwitchKit. It provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

Message Headers in C

Description

To process all of the basic message handling required by the EXS API, SwitchKit presents each message with a header and supplies the appropriate fields for the EXS Message Type, Size, Sequence numbers, and Node ID.

The data portion of the message can be an SwitchKit Message, an outbound (for example host to switch) or inbound (for example switch to host) EXS API message, or an acknowledgment message. In all cases, the manipulation of these messages is accomplished through the macros described later in this section.

Syntax of Basic C Message Header

```
typedef struct {
   XBYTE Size;
   int Tag;
   XBYTE SeqNum;
   UBYTE NodeID;
   int EngineName;
   } BaseFields;
```

Message Structure

To communicate with the LLC, sequences of bytes must be passed through interprocess communication. To hide the underlying byte representation, SwitchKit provides a set of message structures to represent the data more intelligently. These message structures are all defined in the *Messages.api.h* API file. The structures are hierarchical, so that a simple cast can convert one to another.

Example

```
typedef struct {
   unsigned short Span;
   UBYTE Channel;
   UBYTE ResendFlag;
   } XL_RequestForService;

MsgStruct msg;
if (sk_getMsgTag(&msg) == TAG_RequestForService)
{
   XL_RequestForService *rfs;
   rfs = (XL_RequestForService *)&msg;
}
```

Message Structure Macros

Description

To send a message, you must first create a message structure macro.

There are two ways of creating message structure macros. You can choose to use predefined macros included in SwitchKit, or you can write you own macros. Either way, you must use a switch statement to go through the message.

Switch Statement with Predefined Macros

The received message structure includes information describing what kind of message it is. You handle this information with the ***SK_MSG_SWITCH*** macros. These macros mimic a switch statement and allow different branches of code to be executed, depending on the kind of message. Within the branch, a cast is performed for you automatically, allowing you to operate on a variable that has the correct type. The switch statement must end with ***SK_END_SWITCH***.

Example for C with Predefined Macros

```
MsgStruct *msg
SK_MSG_SWITCH(msg)
{
   CASE_ConnectAck(ca)
      return processStatus(ca->Status);
   CASE_ChannelReleased(cr)
   {
      CleanUpChan(cr->Span, cr->Channel);
      break;
   }
   CASE_default
   {
      /*Unexpected msg!*/
      break;
   }
} SK_END_SWITCH
```

In the ***SK_MSG_SWITCH*** statement, “break” can be used normally, and the lack of a break statement causes execution to fall through to the next branch, just as it would in a normal switch statement. However, the default branch must be labeled with ***CASE_default***, as in the above example. The other important syntax difference from a normal switch statement is the lack of colons after the case label. The macro inserts the colons for you.

In the body of the first two cases, the programmer has referenced structure elements, Status, Span, and Channel that are only accessed through a correctly-cast ***MsgStruct***. For example, in the first case *ca* has been defined by the ***CASE_ConnectAck*** macro to be a pointer to a ***ConnectAck*** structure, and is initialized with *msg*.

You can find the definitions of the marcros in the included file *API_Funcs.h*. Those macros are defined in terms of more primitive operations, which you can use instead of an ***SK_MSG_SWITCH*** statement.

Switch Statement without Predefined Macros

These are the steps to translate the example to straight C code.

1. Copy the message structure.
2. Do the switch statement on the tag of the message.
3. For the first case, use ***TAG_ConnectAck*** for the tag of the acknowledgment to the connect message. There is no ***XL_ConnectAck*** structure because its acknowledgement is generic, so the message is cast to an ***XL_AcknowledgeMessage***. It is then stored in the *ca* variable, which is used in the body of the case clause.
4. Use the next case for the ***XL_ChannelReleased*** message. This time, the message is cast to a structure that is put in *cr*.
5. CASE_default is not used for anything in this example.

Example for C without Predefined macros

```
SK_Message *sk_saved_msg = msg;
switch(sk_saved_msg -> getMsgTag())
{
   case(TAG_ConnectAck):
   {
      XL_AcknowledgeMessage *ca =
      (XL_AcknowledgeMessage *)sk_saved_msg;
      return processStatus(ca->Status);
   }
   case(TAG_ChannelReleased):
   {
      XL_ChannelReleased *cr =
      (XL_ChannelReleased *)sk_saved_msg;
      cleanUpChan(cr->Span, cr->Channel);
      break;
   }
   default:
      break;
}
```

Message Macros for C Programmers

Description

The following macros are provided to assist in setting up the message. Some macros have a threadsafe version which is indicated by the skts prefix. Macros do not have return values; they evaluate. The listed parameters are to show expected types.

sk_initMsg

This macro initializes the structure by zeroing the sequence number and size, and by setting the tag. This macro must be called on every message structure.

Syntax

```
void sk_initMsg(MsgStruct *m, int tag);
```

Threadsafe Syntax

```
void skts_initMsg(MsgStruct *m, int tag);
```

Proper Initialization

If the application should fail to call sk_initMsg() before sending the message, the following text will appear on the console and in the application's maintenance log:

```
"**Error:Aug 12 2003 09:30:30: Received message with
unknown engine type205 - is the sender of this message
incorrectly using a pre-3.0 version of SwitchKit? This
version of SwitchKit will not work with any pre-
3.0versions."
```

This message incorrectly identifies the source of the problem but indicates that the message was not properly initialized.

sk_setMsgSize

This macro sets the size of the message. In general, this macro should not be called. For all fixed-size messages, and almost all variably-sized messages, SwitchKit computes the size of the message automatically. If you don't set the size, and SwitchKit cannot compute it, you get this error code returned: `SK_NO_MSG_SIZE`. In this case, the size should be set to the length field that would appear in the EXS message, as described by the *API Reference*. (Note that the actual value set in the MsgStruct.Size field will not necessarily be this exact value.)

Syntax

```
void sk_setMsgSize(MsgStruct *m, aBufSize);
```

Threadsafe Syntax

```
XBYTE skts_setMsgSize(MsgStruct *m);
```

sk_getMsgSize This macro retrieves the size of the message that was previously set via sk_setMsgSize. This function will not cause SwitchKit to recompute the size of the message and therefore should only be called if you are calling sk_setMsgSize(). See sk_setMsgSize() for more details.

Syntax

```
XBYTE sk_getMsgSize(MsgStruct *m);
```

Threadsafe Syntax

```
XBYTE skts_getMsgSize(MsgStruct *m);
```

sk_setRange This macro sets the range of spans and channels for a ChanRangeMsgStruct message. The macro is equivalent to:

```
crmsg.StartSpan = startSpan;
crmsg.StartChan = startChan;
crmsg.EndSpan = endSpan;
crmsg.EndChan = endChan;
```

Syntax

```
void sk_setRange (ChanRangeMsgStruct* crMsg, int
   startSpan, int startChan, int endSpan, int endChan);
```

Threadsafe Syntax

```
void skts_setRange (ChanRangeMsgStruct* crMsg, int
   startSpan, int startChan, int endSpan, int endChan);
```

sk_getMsgType This macro evaluates the following cases:

- SK_TYPE_INBOUND: for messages arriving from the switch.
- SK_TYPE_OUTBOUND: for messages destined for the switch.
- SK_TYPE_TOOLKIT: for SwitchKit-specific messages, for example Add LLCNode.
- SK_TYPE_ADMIN: for SwitchKit administrative messages, for example AssociateChannelGroup.
- SK_TYPE_DUMMY: for SwitchManager-specific messages, for example PPLTool, AllInService.

Syntax

```
int sk_getMsgType(MsgStruct *m);
```

Threadsafe Syntax

```
int skts_getMsgType(MsgStruct *m);
```

isInboundMsg This macro evaluates TRUE if m points to an inbound message.

Syntax

```
bool isInboundMsg(MsgStruct *m);
```

isOutboundMsg This macro evaluates TRUE if m points to an outbound message.

Syntax

```
bool isOutboundMsg(MsgStruct *m);
```

isToolkit This macro evaluates TRUE if m points to a toolkit message.

Syntax

```
bool isToolkit(MsgStruct *m);
```

sk_getXLTag This macro evaluates to the UBYTE message tag as received by the LLC from the switch.

Syntax

```
UBYTE sk_getXLTag(MsgStruct *m);
```

Threadsafe Syntax

```
UBYTE skts_getXLTag(MsgStruct *m);
```

sk_getMsgTag This macro evaluates to the SK tag and will be offset by 10000 if m points to an acknowledgement.

Syntax

```
int sk_getMsgTag(MsgStruct *m);
```

Threadsafe Syntax

```
int skts_getMsgTag(MsgStruct *m);
```

sk_getSeqNum This macro evaluates to the sequence number for the message m is pointing to.

Syntax

```
XBYTE sk_getSegNum(MsgStruct *m);
```

Threadsafe Syntax

```
XBYTE skts_getSegNum(MsgStruct *m);
```

sk_setSeqNum This macro is a convienience macro that sets the sequence number of m to seqNum.

Syntax

```
void sk_setSeqNum(MsgStruct *m, XBYTE seqNum);
```

Threadsafe Syntax

```
void skts_setSeqNum(MsgStruct *m, XBYTE seqNum);
```

sk_is_positive_ack This macro evaluates to TRUE if ack is an inbound message and ack->Status == 0x10.

Syntax

```
bool sk_is_positive_ack(MsgStruct *m);
```

Instantiating a Large SwitchKit Message

Overview

This section describes how to instantiate a SwitchKit message larger than the default size. The methods outlined in this section apply to all SwitchKit messages. When instantiating, the size of the message is defined as the size of the header plus data; not just the size of the message data. For example, if the header is eight bytes and data is 20 bytes, the total message size is 28 bytes.

The two methods for instantiating a message are:

- Accept the default size
- Customize the message size.

C SwitchKit programmers:

To accept the ***default size*** of a message, do one of the following:

```
XL_SampleMessage *sampleMessagePtr = (XL_SampleMessage
   *)malloc(sizeof XL_SampleMessage);
```

Or:

```
XL_SampleMessage sampleMessage;
```

To instantiate a ***custom-sized*** message, do the following:

```
XL_SampleMessage *sampleMessagePtr = (XL_SampleMessage
   *)malloc(1000);
```

Then copy the data into the message (size is the actual size of the data for the message):

```
memcpy (sampleMessagePtr->Data, buf, size);
```

C++ SwitchKit programmers:

To accept the ***default size*** of a message, do one of the following:

```
XLC_SampleMessage *sampleMessagePtr =new
   XLC_SampleMessage;
```

Or

```
XLC_SampleMessage sampleMessage;
```

To instantiate a ***custom-sized*** message, do the following:

```
XLC_SampleMessage *sampleMessagePtr =new
   XLC_SampleMessage(1000);
```

Or

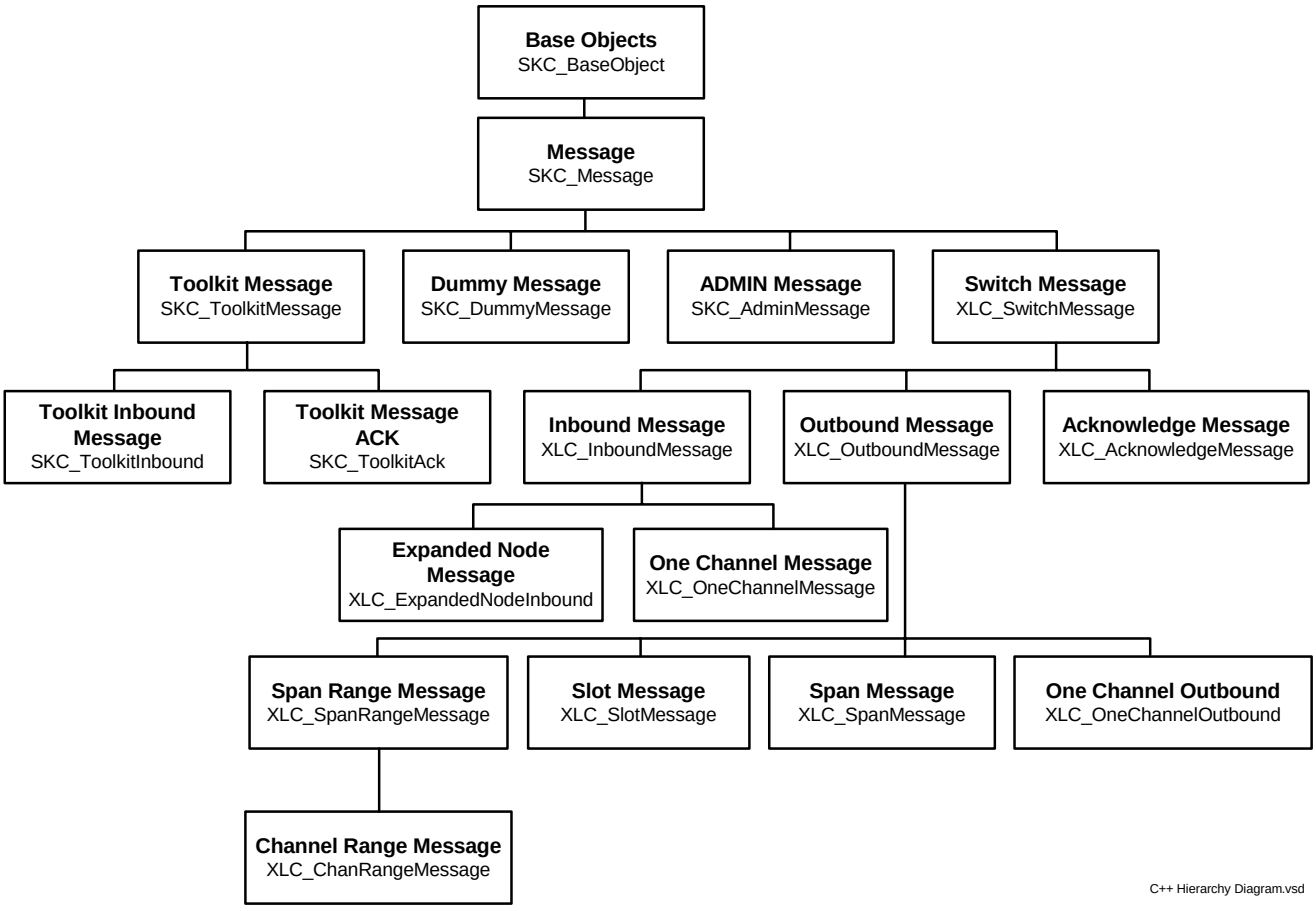
```
XLC_SampleMessage sampleMessage(1000);
```

Then copy the data into the message (where size is the actual size of the data for the message)

```
memcpy (sampleMessagePtr->getDataPtr(), buf, size);
```

Messages in C++

Description The C++ API is a rich class hierarchy that corresponds to the set of EXS and SwitchKit messages that can be sent to the CSP. The class hierarchy lends structure and organization to these messages, as shown in the C++ Class Hierarchy diagram. Macros corresponding to the C macros are provided for the C++ message classes. These macros allow easy branching, based on the type of message received.



SKC_BaseObject This class is the base of the class hierarchy tree. As delivered in the API, it does not derive from any other class. *SKC_BaseObject* implements a basic mechanism for determining if a given object is derived from a particular class. The ***desc***() virtual function returns a static object pointer that is different for each class. By calling ***isKindOf***(), the ***SKC_ClassDesc*** of a given class is compared against the ***desc***() of the given object and all of its base classes. If ***isKindOf***() returns true, then the type cast to the given type is safe.

Example

```
SKC_BaseObject *obj = getObject();
if (obj->isKindOf(&SKC_ToolkitMessage:desc())) {
   SKC_ToolkitMessage *tkitMsg = (SKC_ToolkitMessage *)
   obj;
   processToolkitMessage(tkitMsg);
   }
```

Class Definition Macros

Every class defined in the SwitchKit C++ API contains an invocation of a macro both in its class declaration (the .h file) and in its class definition (the C file). As distributed, these macros complete the implementation of the class derivation testing mechanism.

The macro invoked in the header file is ***SK_DECLARE_CLASS***(cls, base). The first argument is the name of the class that the macro invocation appears in, and the second argument is its base class. ***SK_DEFINE_CLASS***(cls) is invoked in the implementation file of the class—once for each class defined by the API with an argument of that class name.

For example, if you want to add a trivial ***print***() function so that all of the classes know how to print themselves, ***SK_DECLARE_CLASS*** could be appended with the following:

```
#define SK_DECLARE_CLASS(cls,base) \
virtual void print() {cout << “Object of type” << #cls; }
   \
...
```

In the above example, it is assumed that #cls is interpreted by the processor to substitute in the value of the cls argument, enclosed in quotes.

Base Classes

Description The following are the Base Classes for C++.

SKC_Message The SKC_Message class is the base class for all messages. An object of this class knows how to send itself to the switch, and can be asked for its message tag, message type, and message name. Furthermore, a pointer to the MsgStruct that serves as its underlying representation can be returned with a call to getStructPtr().

SKC_ToolkitMessage This class is the base class for all messages that are defined for SwitchKit specific functions.

SKC_DummyMessage This is the base class for the dummy messages. Dummy messages are specific to SwitchManager configuration files and are for internal SwitchKit use only. Do not use dummy messages for your application development.

SKC_AdminMessage This is the base class for messages that perform LLC administrative tasks. Do not use these administrative messages for your application development.

SKC_ToolkitAck This class is the base class for all Toolkit messages that are acknowledgments to a previously sent message.

SKC_TollkitInbound This is the base class for unsolicited messages from the LLC. These are messages that are generated inside the LLC for the benefit of application developers. An example of an SKC_ToolkitInbound message would be SKC_ConnectionStatusMsg.

XLC_SwitchMessage All message classes that correspond to EXS-defined messages are derived from this class. This class defines a notion of message size, which corresponds to the size of an EXS variably-sized message.

XLC_OutboundMessage An XLC_OutboundMessage is an XLC_SwitchMessage that is going out to the switch (that is, a host-initiated message).

XLC_InboundMessage An XLC_InboundMessage is an XLC_SwitchMessage that is coming in from the switch (that is, an CSP-initiated message). It is not an acknowledgment.

XLC_AcknowledgeMessage An XLC_AcknowledgeMessage is an XLC_SwitchMessage that is coming in from the switch in response to a previously sent outbound message.

XLC_ExpandedNodeInbound An XLC_ExpandedNodeInbound is an XLC_Message that includes the Expanded Logical Node ID AIB.

XLC_OneChannelMessage An XLC_OneChannelMessage is an XLC_InboundMessage that is occurring on a specific channel. It defines functions for getting the message's relevant span and channel. An example of this would be a Request For Service message.

XLC_SpanRangeMessage An XLC_SpanRangeMessage is an XLC_OutboundMessage that applies to a range of spans. It defines functions for getting and setting the starting and ending message span. The only current XLC_SpanRangeMessage is XLC_J1SpanConfig.

XLC_ChanRangeMessage An XLC_ChanRangeMessage is an XLC_SpanRangeMessage that not only has a starting and ending span, but also has a starting and ending channel. The message applies to all of the channels between the starting span channel pair and the ending span channel pair. Most configuration messages are XLC_ChanRangeMessages.

XLC_SlotMessage An XLC_SlotMessage is an XLC_OutboundMessage that acts on a specific slot. It defines functions for getting and setting the target slot.

XLC_SpanMessage

An XLC_SpanMessage is an XLC_OutboundMessage that acts on a specific span. It defines functions for getting and setting the target span.

XLC_OneChannel Outbound

An XLC_OneChannelOutbound is an XLC_OutboundMessage that acts on a specific channel. It is equivalent to XLC_OneChannelMessage, except that it applies to host-initiated messages instead of switch-initiated messages. An example of an XLC_OneChannelMessage is XLC_ChannelParameterQuery.

Derived Leaf Message Objects

All leaf message objects are derived from the base classes described above. The definitions of these objects are found in CMessages.api.h. CMessages.api.CPP contains the implementation of these objects. Their constructors all automatically initialize the underlying C structure, so you don't need to call sk_initMsg.

For example, to send a *CardStatusQuery* to the card in Slot 3 through the C++ API, you would enter:

```
XLC_CardStatusQuery csq;
csq.setSlot(3);
csq.send(tag, handlerFunction);
```

Message Class Macros

Description For C++ programmers, SwitchKit provides macros that work with message classes instead of structures.

Syntax The C++ syntax for a basic message is:

```
SKC_Message ***SK_msg***(const MsgStruct *, bool fromSwitch,
   int bufSz = 0);
```

Switch Statement The received message class includes information describing what kind of message it is. You handle this information with the ***SKC_MSG_SWITCH*** macros. These macros mimic a switch statement, and allow different branches of code to be executed, depending on the type of message. Within the branch, a cast is automatically performed for you, allowing you to operate on a variable of the correct type. The switch statement must end with ***SKC_END_SWITCH***.

Example for C++ with Predefined Macros

```
SK_Event *incomingEvent = getEvent();
SKC_Message *msg = incomingEvent->IncomingCMsg;
SKC_MSG_SWITCH(msg)
{
   CASEC_ConnectAck(ca)
      return processStatus(ca->getStatus());
   CASEC_ChannelReleased(cr)
   {
      CleanUpChan(cr->getSpan(), cr->getChannel());
      break;
   }
   CASEC_default
   {
      /*Unexpected msg!*/
      break;
   }
} SKC_END_SWITCH
```

In the ***SKC_MSG_SWITCH*** statement, “break” can be used normally, and the lack of a break statement causes execution to fall through to the next branch, just as a normal switch statement would. However, the default branch must be labeled with ***CASEC_default***, as in the above example. The other important syntax difference from a normal switch statement is the lack of colons after the case label. The macro inserts the colons for you.

You can find the definitions of the marcros in the included file *API_Funcs.h*. Those macros are defined in terms of more primitive operations, which you can use instead of an ***SKC_MSG_SWITCH*** statement.

Steps to Translate Example to Straight C++ Code

1. Copy the message class.
2. Do the switch statement on the tag of the message.
3. For the first case, use TAG_ConnectAck for the tag of the acknowledgment to the connect message. There is no *XLC_ConnectAck* class because its acknowledgement is generic, so the message is cast to an *XLC_AcknowledgeMessage*. It is then stored in the **ca** variable, which is used in the body of the case clause.
4. Use the next case for the *XLC_ChannelReleased* message. This time, the message is cast to a class that is put in **cr**.
5. You don't use the default clause for anything in this example.

Example for C without Predefined macros

```
SKC_Message *sk_saved_msg = msg;
switch(sk_saved_msg -> getMsgTag())
{
   case(TAG_ConnectAck):
   {
      XLC_AcknowledgeMessage *ca =
      (XLC_AcknowledgeMessage *)sk_saved_msg;
      return processStatus(ca->getStatus());
   }
   case(TAG_ChannelReleased):
   {
      XLC_ChannelReleased *cr =
      (XLC_ChannelReleased *)sk_saved_msg;
      cleanUpChan(cr->getSpan, cr->getChannel);
      break;
   }
   default:
      break;
}
```

sk_packAutoStorage()

Description The ***sk_packAutoStorage***() function converts the message into a sequence of bytes suitable for sending to the LLC. This function is equivalent to ***sk_packMessage***(), except that instead of requiring you to pass in a pre-allocated buffer, it automatically allocates an appropriate buffer with sufficient storage for the given message.

Syntax

```
int sk_packAutoStorage(MsgStruct *aMsgStruct, char
   **BufferPtr, int *aBufSize);
```

Parameters The function parameters are shown in the table below.

Argument	Description
MsgStruct **m	Pointer to a generic C Structure representing a message.
aBufferPtr	Pointer to a packed message pointing to memory allocated and owned by SwitchKit API. Application wishing to retain this information beyond the next call to a SwitchKit API function should copy the data to memory allocated and controlled by the application as the memory may be reused the next time a SwitchKit API function is called.
aBufSize	Size of the packed message.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_UNKNOWN_MSG_TYPE	The tag field of this MsgStruct is invalid. Did you call ***sk_initMsg***()?

SK_BAD_MSG_SIZE	The size field computed by SwitchKit does not match the size field set through ***sk_setMsgSize***(). For this message, you should not set the message size, and instead should let SwitchKit compute it automatically.
SK_NO_MSG_SIZE	For this MsgStruct, no size was set in with ***sk_setMsgSize***(), and its size couldn't be computed.

sk_packMessage()

Description The ***sk_packMessage***() function converts the message into a sequence of bytes suitable for sending to the LLC. This sequence of bytes is suitable for ***sk_sendMessage***(), ***sk_sendMessageWithTag***(), or ***sk_sendMessageWithHandler*****()**.

Syntax

```
int sk_packMessage(MsgStruct *aMsgStruct, char *aBuffer,
   int *aBufSize);
```

Parameters The function parameters are shown in the table below.

Argument	I/O	Description
MsgStruct **m	I	Pointer to a generic C Structure representing a message.
aBuffer	O	Pointer to a packed message.
aBufSize	I	The size of aBuffer. This tells the SwitchKit API how large the user-provided buffer is and allows the API to stay within the provided buffer.
	O	The number of bytes of aBuffer that were actually used to store the packed message.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_UNKNOWN_MSG_TYPE	The tag field of this MsgStruct is invalid. Did you call ***sk_initMsg***()?
SK_BAD_MSG_SIZE	The size field computed by SwitchKit does not match the size field set through ***sk_setMsgSize***(). For this message, you should not set the message size, and instead should let SwitchKit compute it automatically.

SK_NO_MSG_SIZE	For this MsgStruct, no size was set in with ***sk_setMsgSize***(), and its size couldn't be computed.

sk_sendMessage()

Description The ***sk_sendMessage***() function takes the results of ***sk_packMessage***() and sends a character buffer with a size to the switch. If no appropriate message structure is available to pass to ***sk_packMessage***(), the character buffer can be filled in manually. This message is also available as ***sk_sendMessageOnConnection()*** and ***skts_sendMessage()***.

Syntax

```
int sk_sendMessage(char *aBuffer, aBufSize);
int sk_sendMessageOnConnection(char *aBuffer, aBufSize,
   int aConID);
```

Parameters The function parameters are shown in the table below

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_UNKNOWN_MSG_BUF	Buffer and Size pointers do not point to a valid message.

SK_NO_CONNECT_AVAIL	The specified connection from OnConnection() is not available or valid.

sk_sendMessageWithHandler()

Description

The ***sk_sendMessageWithHander***() function takes the results of ***sk_packMessage***() and sends a character buffer indicating the size to the switch. If no appropriate message structure is available to pass to ***sk_packMessage***(), the character buffer can be filled in manually.

This function takes a void pointer called tag as an argument. It can be an arbitrary value and is not used or change by SwitchKit. Upon receipt of an acknowledgment, the value is passed to the ***HandlerFunc*** to provide some context associated with the sent message.

The ***sk_sendMessageWithHandler()*** function also takes a pointer to ***HandlerFunc*** as an argument. The function ***HandlerFunc*** points to is called to handle the receipt of any acknowledgment, when or if it arrives.

This message is also available as ***sk_sendMessageWithHandlerOnConnection()***.

Syntax

```
int sk_sendMessageWithHandler(char *aBuffer, aBufSize,
   void *aTag, HandlerFunc *);
int sk_sendMessageWithHandlerOnConnection(char *aBuffer,
   int sz, void *aTag, HandlerFunc *aHandlerFunc, int
   aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_UNKNOWN_MSG_BUF	Buffer and Size pointers do not point to a valid message.
SK_NO_CONNECT_AVAIL	The specified connection from OnConnection() is not available or valid.

sk_sendMessageWithTag()

Description

The ***sk_sendMessageWithTag***() function takes the results of ***sk_packMessage***() and sends a character buffer with a size to the switch. If no appropriate message structure is available to pass to ***sk_packMessage***(), the character buffer can be filled in manually with the bytes that need to be sent.

This function takes a void pointer called tag as an argument. It can be an arbitrary value and is not used or change by SwitchKit. Upon receipt of an acknowledgment, the value is passed to the ***HandlerFunc*** to provide some context associated with the sent message.

This message is also available as ***sk_sendMessageWithTagOnConnection()***.

Syntax

```
int sk_sendMessageWithTag(char *aBuffer, aBufSize,
   void *aTag);
int sk_sendMessageWithTagOnConnection(char *aBuffer,
   aBufSize, void *aTag, int aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.

SK_UNKNOWN_MSG_BUF	Buffer and Size pointers do not point to a valid message.
SK_NO_CONNECT_AVAIL	The specified connection from OnConnection() is not available or valid.

sk_sendMsgStruct()

Description

The ***sk_sendMsgStruct()*** function is a convenience function that first packs a message, and then calls ***sk_sendMessageWithHandler().*** This message is also available as ***sk_sendMsgStructOnConnection()*** and ***skts_sendMsgStruct()***.

Syntax

```
int sk_sendMsgStruct(MsgStruct *aMsgStruct, void *aTag,
   HandlerFunc *aHandlerFunc);
int sk_sendMsgStructOnConnection(MsgStruct *aMsgStruct,
   void *aTag, HandlerFunc *aHandlerFunc, int aConID);
int skts_sendMsgStructOnConnection(MsgStruct *aMsgStruct,
   void *aTag, HandlerFunc *aHandlerFunc, int aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
*aMsgStruct	Pointer to a generic C Structure representing a message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values

Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_UNKNOWN_MSG_TYPE	The tag field of this MsgStruct is invalid. Did you call sk_initMsg()?

SK_UNKNOWN_MSG_BUF	Buffer and Size pointers do not point to a valid message.
SK_NO_CONNECT_AVAIL	The specified connection from OnConnection() is not available or valid.

sk_unpackAutoStorage()

Description The ***sk_unpackAutoStorage***() function takes the character buffer and size returned by sk_rcvMessage() and unpacks it into a MsgStruct. This function is equivalent to ***sk_unpackMessage***(), except that instead of requiring you to pass in a pre-allocated buffer, it automatically allocates an appropriate buffer with sufficient storage for the given message.

Syntax

```
int sk_unpackAutoStorage(char *aBuffer, int aBufSize,
   MsgStruct *aMsgStructPtr);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
*aMsgStructPtr	Pointer to a generic C Structure representing a message pointing to memory allocated and owned by SwitchKit API.

Important! The storage of this buffer is managed automatically by SwitchKit.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK__MSG_TOO_BIG	The amount of memory needid to be allocated is greater than 65536 bytes.
SK_INTERNAL_ERROR	Message could not be unpacked.

sk_unpackMessage()

Description The ***sk_unpackMessage***() function takes the character buffer and size returned by ***sk_rcvMessage***() and unpacks it into a message structure.

Syntax

```
int sk_unpackMessage(char *aBuffer, int aBufSize,
   MsgStruct *aMsgStruct);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aMsgStruct	Pointer to a generic C Structure representing a message.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_INTERNAL_ERROR	The message could not be unpacked.

sk_rcvAndDispatch()

Description

The ***sk_rcvAndDispatch***() is a replacement for ***sk_rcvMessage***() that knows about handler functions. If you are using handler functions anywhere in your code, always call ***sk_rcvAndDispatch***(), or the messages will not be dispatched correctly. When a message arrives, ***sk_rcvAndDispatch***() determines whether there is any handler function associated with that message. If so, it calls the handler with the appropriate arguments.

sk_rcvAndDispatch() calls the handler function with any tag received and, if this message is an acknowledgment, it also passes in a pointer to the original MsgStruct that triggered this acknowledgment. Notice that all messages passed to the handler function are already unpacked. ***sk_rcvAndDispatch***() returns whatever integer the handler function returns. Because the main ***sk_rcvAndDispatch***() loop checks the return values, it is important that all of the handler functions return a valid integer. Any integer may be returned. Since many SwitchKit #defines use negative integers for many error codes, for example SK_LOST_LLC is -3, you should restrict yourself to returning ***non-negative*** integers only.

If there is no associated handler function, rcvAndDispatch returns `SK_NOT_HANDLED`, and its buffer and size are indicated, just as though ***sk_rcvMessage***() had been called. The application module can then unpack and handle the undispatched message.

This message is also available as ***sk_rcvAndDispatchOnConnection()*** and ***skts_rcvAndDispatch()***.

Syntax

```
int sk_rcvAndDispatch(char *aBuffer, int *aBufSize,
   double aTimeout, void **aTag);
int sk_rcvAndDispatchOnConnection(char *aBuffer, int
   *aBufSize, double aTimeout, void **aTag, int
   *aConIDPtr);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aTimeout	The maximum time the function should wait for a message before returning. The timeout value is a floating point number allow the application to specify a fractional portion to the timeout. For example, a timeout value of 1.5 seconds would be interpreted as a one and one-half second timeout. If no message arrives which is destined for the application in the specified time, the function should return SK_NO_MESSAGE.
aTag	The application-defined pointer that is only valid if the function returns SK_NOT_HANDLED. The value was set in either the original message send operation if the message is an acknowledgement or was set when the handler was defined if the message is switch-initiated.
aConIDPtr	This is used to identify from which LLC connection the message was received.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_NO_MESSAGE	No message was received. The time-out expired before any messages (other then Poll messages) were received by the LLC.
SK_BAD_MESSAGE	An improperly formatted message was received.
SK_NOT_HANDLED	Message was not handled by handler function, no results available.
SK_MSG_TOO_BIG	This value is returned because the buffer for a message was too small, or because the message was larger than the limit (64 KB). The size integer value that you pass in is an input/output variable that indicates the size of the message that it unpacked, regardless of what you passed it.
SK_INVALID_LINK	A message was directed to a node that LLC was no longer connected to. This could happen when a connection to a node is unexpectedly severed.
SK_FRAMING_FAILURE	Message being sent to a node has a mismatched length between the actual length and the calculated length Reformat the message and send again.
SK_SOCKET_WRITE_FAILURE	LLC encountered an issue writing a message to a socket. This could be the result of a full socket (the distant end not reading from the socket appropriately) or a severed connection.

SK_NODE_NOT_FOUND	A message was directed to a node that LLC is not connected to. This could happen when an application specifies a node ID that has not been added to this configuration.

Example Code sample demonstrating the use of ***sk_rcvAndDispatch***().

```
#define MSG_SIZE=1024
main(int argc, char *argv[])
{
   char buf[MSG_SIZE];
   int sz, ret;
   void *dta;
   char * AppName;
   SKC_Message * msg;
...
/*Initialize connection to LLC with name*/
sk_initializeConnection(TANDEM_APP);
...
   while (1) {
      sz = MSG_SIZE;
      /*Initialize maximum size of message to receive*/

      if (debug > 5)

      printf("\nWaiting for an Inseize at %s for 600
    secs...\n", argv[1]);

      /* Wait up to 600 seconds for a message.  If a
    message is received that has handler function(s),
    rcvAndDispatch will automatically call the handler
    function(s) and return the result of the last handler
    function called.  If there are no handler functions,
    then SK_NOT_HANDLED is returned, and we just print
    some info about the message.*/

      ret = sk_rcvAndDispatch(buf, &sz, 600, &dta);
      ....
      } /* (ret == SK_NOT_HANDLED)   */
   ...
   } /* while (1) */
} /* main */
```

sk_rcvAndDispatchAutoStorage()

Description The ***sk_rcvAndDispatchAutoStorage***() is identical to ***sk_rcvAndDispatch***(), except that instead of passing in a pre-allocated buffer, this function automatically allocates enough memory to receive whatever message comes from the switch. This message is also available as ***sk_rcvAndDispatchAutoStorageOnConnection()*** and ***skts_rcvAndDispatchAutoStorage()***.

Syntax

```
int sk_rcvAndDispatchAutoStorage(char **aBufferPtr, int
   *aBufSize, double aTimeout, void **notCurrentlyUsed);
int sk_rcvAndDispatchAutoStorageOnConnection(char
   **aBufferPtr, int *aBufSize, double aTimeout, void
   **notCurrentlyUsed, int *aConIDPtr);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBufferPtr	Pointer to a packed message pointing to memory allocated and owned by SwitchKit API. Application wishing to retain this information beyond the next call to a SwitchKit API function should copy the data to memory allocated and controlled by the application as the memory may be reused the next time a SwitchKit API function is called.
aBufSize	Size of the packed message.
aTimeout	The maximum time the function should wait for a message before returning. The timeout value is a floating point number allow the application to specify a fractional portion to the timeout. For example, a timeout value of 1.5 seconds would be interpreted as a one and one-half second timeout. If no message arrives which is destined for the application in the specified time, the function should return SK_NO_MESSAGE.
NotCurrentlyUsed	The field is no longer used by the SwitchKit API and still exists for backward compatibility. This value should be set to NULL(0).
aConIDPtr	This is used to identify from which LLC connection the message was received.

Important! The storage of this buffer is managed automatically by SwitchKit.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_NO_MESSAGE	No message was received. The time-out expired before a message was received or the LLC received a poll message from the switch.
SK_BAD_MESSAGE	An improperly formatted message was received
SK_NOT_HANDLED	Message was not handled by handler function, and no results are available.
SK_INVALID_LINK	A message was directed to a node that LLC was no longer connected to. This could happen when a connection to a node is unexpectedly severed.
SK_FRAMING_FAILURE	Message being sent to a node has a mismatched length between the actual length and the calculated length Reformat the message and send again.
SK_SOCKET_WRITE_FAILURE	LLC encountered an issue writing a message to a socket. This could be the result of a full socket (the distant end not reading from the socket appropriately) or a severed connection.

SK_NODE_NOT_FOUND	A message was directed to a node that LLC is not connected to. This could happen when an application specifies a node ID that has not been added to this configuration.

sk_rcvMessage()

Description

The ***sk_rcvMessage***() function tries to receive a message from the switch. The buffer argument must be at least 256 bytes long. The buf argument gets filled in with the message, and the size argument is set to its size. The timeout argument specifies how long to wait for a message, in seconds. A value less than 0 means to wait indefinitely. A value of 0 specifies a polling behavior, returning immediately whether or not there is a message. A value greater than 0 will wait for a maximum of that many seconds, or until a message arrives. The actual accuracy of the timeout timer varies across machines.

> **Important!** Setting your timeout argument to a value of zero (0) or smaller might cause a significant increase in CPU usage.

The ***sk_rcvMessage***() function also looks for any tag data. If the received message is an acknowledgment to a previously sent message, then *dta is set to whatever tag was sent with the original message. If no tag was sent, or if the message was a switch-originated message instead of an acknowledgment, then *dta is assigned 0.

This message is also available as ***sk_rcvMessageOnConnection()***.

Syntax

```
int sk_rcvMessage(char *aBuffer, int *aBufSize, double
   aTimeout, void **aDataPtr);
int sk_rcvMessageOnConnection(char *aBuffer, int
   *aBufSize, double aTimeout, void **aDataPtr, int
   *aConID);
```

Parameters

The function parameters are shown in the next table.

Argument	Description
aBuffer	Pointer to a packed message.
aBufSize	Size of the packed message.
aTimeout	The maximum time the function should wait for a message before returning. The timeout value is a floating point number allow the application to specify a fractional portion to the timeout. For example, a timeout value of 1.5 seconds would be interpreted as a one and one-half second timeout. If no message arrives which is destined for the application in the specified time, the function should return SK_NO_MESSAGE.
aDataPtr	This argument is not currently used.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with.

Return Values Possible return values for this function:

SK_LOST_LLC	This return value indicates that your application lost contact with the LLC.
SK_NO_MESSAGE	No message was received. The time-out expired before a message was received or the LLC received a poll message from the switch.
SK_BAD_MESSAGE	An improperly formatted message was received
SK_INVALID_LINK	A message was directed to a node that LLC was no longer connected to. This could happen when a connection to a node is unexpectedly severed.

SK_FRAMING_FAILURE	Message being sent to a node has a mismatched length between the actual length and the calculated length Reformat the message and send again.
SK_SOCKET_WRITE_FAILURE	LLC encountered an issue writing a message to a socket. This could be the result of a full socket (the distant end not reading from the socket appropriately) or a severed connection.
SK_NODE_NOT_FOUND	A message was directed to a node that LLC is not connected to. This could happen when an application specifies a node ID that has not been added to this configuration.

8 Utility Functions

Purpose This chapter describes the utility functions available in SwitchKit. It provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions. Some functions return pointers; those functions do not have a status value indicating success or failure.

sk_extractExtendedICBFromChannelReleasedWithData()

Overview The ***sk_extractExtendedICBFromChannelReleasedWithData()*** function is used to extract ICBSubType, ICBLength & ICBData from an SK_ChannelReleasedWithData message independent of the ICBType (Data Type or Extended Data Type). This function compensates for the fact that the ICBSubType and ICBLength fields may be two bytes each for messages with the Extended Data Type ICB or one byte each for all other ICB's. All developers extracting information from an SK_ChannelReleaseWithData message should use this function rather than using the fields defined in the C Structure. Using this function prevents any future incompatibilities with other types of ICB data stored in this message. This function is also available as ***skts_extractExtendedICBFromChannelReleasedWithData()***.

Syntax

```
sk_extractExtendedICBFromChannelReleasedWithData(int*
   xtICBSubtype,int* xtICBLength,UBYTE** xtICBData,
   XL_ChannelReleasedWithData* crwdMsg)
```

Threadsafe Syntax

```
int skts_extractExtendedICBFromChannelReleasedWithData(
   int* xtICBSubtype, int* xtICBLength, UBYTE**
   xtICBData, XL_ChannelReleasedWithData* crwdMsg);
```

Parameters The function parameters are shown in the table below.

Argument	Description
xtICBSubtype	ICBSubtype value will be returned in this output argument.
xtICBLength	ICBLength will be returned in this output argument.
xtICBData	A pointer to the ICBData will be returned in this output argument.
crwdMsg	The ChannelReleasedWith Data message from which the above- mentioned output is to be extracted.

Return Value The return value for this function is always:

```
TRUE(1)
```

sk_getManagedFile()

Description

This function allows you to access the internal file management capabilities. It returns a pointer to a FILE structure of an opened file that is ready for writing. You can write any data to this file.

The benefits of using managed files are:

- the location of the file is determined by SK_LOG_DIR.
- the file is rotated based on the same criteria as other SwitchKit logs, for example hourly, daily and so on.

This function is also available as ***skts_getManagedFile()***.

Syntax

```
FILE *sk_getManagedFile(const char *aFileName);
```

Threadsafe Syntax

```
FILE* skts_getManagedFile(const char *aFileName );
```

Parameters

The function parameters are shown in the table below.

Argument	Description
aFileName	Pointer to specified file.

Return Values

Possible return values for this function are:

SK_LOST_LLC — This return value indicates that your application lost contact with the LLC.

sk_getMessageText()

Description This function converts any MsgStruct into a text string. This function is also available as ***skts_getMessageText()***.

Syntax

```
int sk_getMessageText(MsgStruct *aMsgStruct, char
   *aBuffer);
```

Important! The `char *aBuffer` must be previously allocated and at least 5000 bytes long.

Threadsafe Syntax

```
int skts_getMessageText( MsgStruct *aMsgStruct, char
   *aBuffer );
```

Parameters The function parameters are shown in the table below.

Argument	Description
aMsgStruct	Pointer to a generic C Structure representing a message.
aBuffer	Pointer to a packed message.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_getMsgName()

Description Returns the name of the given message.This function is also available as ***skts_getMsgName()***.

Syntax

```
char* sk_getMsgName(MsgStruct *aMsgStruct);
```

Threadsafe syntax

```
char* skts_getMsgName(MsgStruct *aMsgStruct);
Parameters
```

The function parameters are shown in the table below.

Argument	Description
aMsgStruct	Pointer to a generic C Structure representing a message.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_getMsgSizeFromTag()

Description Returns the size, in bytes, of the C structure associated with a message of type tag. For example, ***sk_getMsgSizeFromTag***(0xa8) would return the size of XL_AssignSpan structure. This function is also available as ***skts_getMsgSizeFromTag()***.

Syntax

```
int sk_getMsgSizeFromTag(int aMessageTag);
```

Threadsafe Syntax

```
int skts_getMsgSizeFromTag( int aMessageTag);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aMessageTag	Uniquely identifies the message. Valid message tags are defined in Messages.api.h and begin TAG_. Each message has a message tag. For example the the RequestForService message, the message tag is TAG_RequestForService.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_getVersionMajor() / sk_getVersionMinor() / sk_getVersionBuild()/ sk_getVersionRelease()

Purpose

Use the ***sk_getVersion***...() function to retrieve SwitchKit version information.

Description

The SwitchKit API version information is divided into several fields that are retrievable using these functions. The SwitchKit API version follows the format: AA.BB.CC.DD.

- AA represents the SwitchKit API major version number.
- BB represents the SwitchKit API minor version number.
- CC represents the SwitchKit API release number.
- DD represents the SwitchKit API build number.

An example of a SwitchKit API version is 8.02.02.35.

Syntax

```
int sk_getVersionMajor();
int sk_getVersionMinor();
int sk_getVersionRelease();
int sk_getVersionBuild();
```

Threadsafe Syntax

```
int skts_getVersionMajor();
int skts_getVersionMinor();
int skts_getVersionRelease();
int skts_getVersionBuild();
```

sk_statusText()

Description Returns a text string associated with status values. This status value can be either the status returned in an EXS message, or the status returned from a SwitchKit function call. This function is also available as ***skts_statusText()***.

Syntax

```
char *sk_statusText(int anError)
```

Threadsafe Syntax

```
char *skts_statusText(int anError)
```

Parameters The function parameters are shown in the table below.

Argument	Description
anError	The status number to convert

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_unparseAlarm()

Description This function converts an XL_Alarm message into a text string, suitable for display. This function is also available as ***skts_unparseAlarm()***.

Syntax

```
int sk_unparseAlarm(char *aBuffer, XL_Alarm *anAlarmMsg);
```

Threadsafe Syntax

```
int skts_unparseAlarm( char *aBuffer, XL_Alarm
   *anAlarmMsg);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
anAlarmMsg	The Alarm that is to be unparsed.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_unparseAlarmCleared()

Description This function converts an *XL_AlarmCleared* message into a text string, suitable for display. This function is also available as ***skts_unparseAlarm()***.

Syntax

```
int sk_unparseAlarmCleared(char *aBuffer, XL_AlarmCleared
   *anAlarmClearedMsg);
```

Threadsafe Syntax

```
int skts_unparseAlarmCleared( char *aBuffer,
   XL_AlarmCleared *anAlarmClearedMsg);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
anAlarmClearedMsg	The AlarmCleared message that is to be unparsed.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_unparsePPLEventIndication()

Description This function converts an *XL_PPLEventIndication* message into a text string, suitable for display. This function is also available as ***skts_unparsePPLEventIndication()***.

Syntax

```
int sk_unparseEventIndication(char *aBuffer,
   XL_PPLEventIndication *aPPLEIMsg);
```

Threadsafe Syntax

```
int skts_unparsePPLEventIndication( char *aBuffer,
   XL_PPLEventIndication *aPPLEIMsg);
```

Parameters The function parameters are shown in the table below.

Argument	Description
aBuffer	Pointer to a packed message.
aPPLEIMsg	The PPLEventIndication message that is to be unparsed.

Return Values Possible return values for this function are:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

9 Addressing Functions

Purpose This chapter describes the Address Information Block (AIB) helper functions available in SwitchKit. It provides a description of what the function does, the syntax and the type of AIB used with a given function.

With addressing functions, "***sk_get****..."* functions return values that are being retrieved. Values are not returned for "***sk_set****...*" functions.

AIB Manipulation Functions

Description

These functions are designed to provide assistance to the application developer in initializing the information contained in various AIBs as described in the *API Reference*.

How Applications use AIB Functions

You must use the appropriate functions for initializing the AIBs. Using the wrong functions or not using all the functions necessary to completely initialize an AIB may result in an improperly defined AIB. See *Table , AIB Manipulation Functions (9-3)* and *Table , SS7 Addressing Functions (9-7)* for a mapping of the functions and AIBs. When attempting to initialize a Range of Channels (0x0D) AIB, you must call all of the following functions:

- setAIBStartSpan()
- setAIBStartChannel()
- setAIBEndSpan()
- setAIBEndChannel()

Addressing Functions

Description To assist with the standard state machines, the SwitchKit API allows for incoming messages to be automatically dispatched to an application

AIB Manipulation Functions

Name	Syntax	Threadsafe Syntax	AIB used
getAIBSize	`int sk_getAIBSize (UBYTE *AIBBlock);`	`int skts_getAIBSIze (UBYTE *AIBBlock);`	All
setAIBRange	`void sk_setAIBRange( UBYTE *AIBBlock, XBYTE aStartSpan, UBYTE aStartChannel, XBYTE anEndSpan, UBYTE anEndChan);`	`void skts_setAIBRange( UBYTE *AIBBlock, XBYTE aStartSpan, UBYTE aStartChannel, XBYTE anEndSpan, UBYTE anEndChan );`	0x0D Range of Channels
setAIBChannel	`void sk_setAIBChannelEntity (UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel);`	`void skts_setAIBChannelEntit y (UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel);`	0x0D Channel
getAIBStartSpan	`XBYTE sk_getAIBStartSpan (const UBYTE *AIBBlock);`	`XBYTE skts_getAIBStartSpan(co nst UBYTE *AIBBlock);`	0x0D Range of Channels
setAIBStartSpan	`void sk_setAIBStartSpan( UBYTE *AIBBlock, XBYTE aStartSpan);`	`void skts_setAIBStartSpan( UBYTE *AIBBlock, XBYTE aStartSpan);`	0x0D Range of Channels
getAIBStartChannel	`UBYTE sk_getAIBStartChannel( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBStartChannel( const UBYTE *AIBBlock );`	0x0D Range of Channels
setAIBStartChannel	`void sk_setAIBStartChannel( UBYTE *AIBBlock, UBYTE aStartChannel );`	`void skts_setAIBStartChannel ( UBYTE *AIBBlock, UBYTE aStartChannel );`	0x0D Range of Channels
getAIBEndSpan	`XBYTE sk_getAIBEndSpan( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBEndSpan( const UBYTE *AIBBlock );`	0x0D Range of Channels

Name	Syntax	Threadsafe Syntax	AIB used
setAIBEndSpan	`void sk_setAIBEndSpan( UBYTE *AIBBlock, XBYTE anEndSpan );`	`void skts_setAIBEndSpan( UBYTE *AIBBlock, XBYTE anEndSpan )`	0x0D Range of Channels
getAIBEndChannel	`UBYTE sk_getAIBEndChannel( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBEndChannel( const UBYTE *AIBBlock );`	0x0D Range of Channels
setAIBEndChannel	`void sk_setAIBEndChannel( UBYTE *AIBBlock, UBYTE anEndChannel);`	`void skts_setAIBEndChannel( UBYTE *AIBBlock, UBYTE anEndChannel);`	0x0D Range of Channels
getAIBSpanA	`XBYTE sk_getAIBSpanA(const UBYTE *AIBBlock );`	`XBYTE skts_getAIBSpanA(const UBYTE *AIBBlock );`	0x0D Channels with two AEs
setAIBSpanA	`void sk_setAIBSpanA( UBYTE *AIBBlock, XBYTE aSpanA );`	`void skts_setAIBSpanA( UBYTE *AIBBlock, XBYTE aSpanA );`	0x0D Channels with two AEs
getAIBChannelA	`UBYTE sk_getAIBChannelA( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBChannelA( const UBYTE *AIBBlock );`	0x0D Channels with two AEs
setAIBChannelA	`void sk_setAIBChannelA( UBYTE *AIBBlock UBYTE aChannelA);`	`void skts_setAIBChannelA( UBYTE *AIBBlock, UBYTE aChannelA );`	0x0D Channels with two AEs
getAIBSpanB	`XBYTE sk_getAIBSpanB( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBSpanB( const UBYTE *AIBBlock );`	0x0D Channels with two AEs
setAIBSpanB	`void sk_setAIBSpanB( UBYTE *AIBBlock, XBYTE aSpanB);`	`void skts_setAIBSpanB( UBYTE *AIBBlock, XBYTE aSpanB);`	0x0D Channels with two AEs

Name	Syntax	Threadsafe Syntax	AIB used
getAIBChannelB	`UBYTE sk_getAIBChannelB( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBChannelB( const UBYTE *AIBBlock);`	0x0D Channels with two AEs
setAIBChannelB	`void sk_setAIBChannelB( UBYTE *AIBBlock, UBYTE aChannelB );`	`void skts_setAIBChannelB( UBYTE *AIBBlock, UBYTE aChannelB );`	0x0D Channels with two AEs
getAIBSpan	`XBYTE sk_getAIBSpan( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBSpan( const UBYTE *AIBBlock );`	0x0C Logical Span ***OR*** 0x0D Channel ***OR*** 0x3C Number of CICs
setAIBSpan	`void sk_setAIBSpan( UBYTE *AIBBlock, XBYTE aSpan );`	`void skts_setAIBSpan( UBYTE *AIBBlock, XBYTE aSpan);`	0x0C Logical Span ***OR*** 0x0D Channel ***OR*** 0x3C Number of CICs
getAIBChannel	`UBYTE sk_getAIBChannel( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBChannel( const UBYTE *AIBBlock);`	0x0D Channel ***OR*** 0x3C Number of CICs

Name	Syntax	Threadsafe Syntax	AIB used
setAIBChannel	`void sk_setAIBChannel( UBYTE *AIBBlock," UBYTE aChannel);`	`void skts_setAIBChannel( UBYTE *AIBBlock, UBYTE aChannel);`	0x0D Channel ***OR*** 0x3C Number of CICs
getAIBNumChannels	`UBYTE sk_getAIBNumChannels( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBNumChannels( const UBYTE *AIBBlock );`	0x3C Number of CICs
setAIBNumChannels	`void sk_setAIBNumChannels( UBYTE *AIBBlock, UBYTE aNumChannels);`	`void skts_setAIBNumChannels( UBYTE *AIBBlock, UBYTE aNumChannels);`	0x3C Number of CICs
getAIBSlot	`UBYTE sk_getAIBSlot( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBSlot( const UBYTE *AIBBlock );`	0x01 Slot
setAIBSlot	`void sk_setAIBSlot( UBYTE *AIBBlock, UBYTE aSlot );`	`void skts_setAIBSlot( UBYTE *AIBBlock, UBYTE aSlot );`	0x01 Slot
setAIBDSPChip	`void sk_setAIBDSPChip( UBYTE *AIBBlock, UBYTE x);`	`void skts_setAIBDSPChip( UBYTE *AIBBlock, UBYTE x);`	0x22 DSP Chip
getAIBDSPSlot	`UBYTE sk_getAIBDSPSlot( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBDSPSlot( const UBYTE *AIBBlock );`	0x12 DSP SIMM
setAIBDSPSlot	`void sk_setAIBDSPSlot( UBYTE *AIBBlock, UBYTE aDSPSlot );`	`void skts_setAIBDSPSlot( UBYTE *AIBBlock, UBYTE aDSPSlot );`	0x12 DSP SIMM
getAIBDSPSIMM	`UBYTE sk_getAIBDSPSIMM( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBDSPSIMM( const UBYTE *AIBBlock );`	0x12 DSP SIMM
setAIBDSPSIMM	`void sk_setAIBDSPSIMM( UBYTE *AIBBlock, UBYTE aDSPSIMM );`	`void skts_setAIBDSPSIMM( UBYTE *AIBBlock, UBYTE aDSPSIMM );`	0x12 DSP SIMM
getAIBSIMM	`UBYTE sk_getAIBSIMM( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBSIMM( const UBYTE *AIBBlock );`	0x12 DSP SIMM

Name	Syntax	Threadsafe Syntax	AIB used
setAIBSIMM	`void sk_setAIBSIMM( UBYTE *AIBBlock, UBYTE aSIMM );`	`void skts_setAIBSIMM( UBYTE *AIBBlock, UBYTE aSIMM );`	0x12 DSP SIMM
getAIBDS3	`UBYTE sk_getAIBDS3( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBDS3( const UBYTE *AIBBlock );`	0x32 DS3 Offset
setAIBDS3	`void sk_setAIBDS3( UBYTE *AIBBlock, UBYTE aDS3 );`	`void skts_setAIBDS3( UBYTE *AIBBlock, UBYTE aDS3 );`	0x32 DS3 Offset
getAIBV5ID	`XBYTE sk_getAIBV5ID( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBV5ID( const UBYTE *AIBBlock );`	0x2C V5 ID
setAIBV5ID	`void sk_setAIBV5ID( UBYTE *AIBBlock, XBYTE aV5ID );`	`void skts_setAIBV5ID( UBYTE *AIBBlock, XBYTE aV5ID );`	0x2C V5 ID
getAIBRouterHandle	`XBYTE sk_getAIBRouterHandle( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBRouterHandle ( const UBYTE *AIBBlock );`	0x29 Router
setAIBRouterHandle	`void sk_setAIBRouterHandle( UBYTE *AIBBlock, XBYTE aRouterHandle );`	`void skts_setAIBRouterHandle ( UBYTE *AIBBlock, XBYTE aRouterHandle );`	0x29 Router
getAIBChannelSlot	`UBYTE sk_getAIBChannelSlot(const UBYTE *AIBBlock);`	`UBYTE skts_getAIBChannelSlot( const UBYTE *AIBBlock);`	0x01 Slot
setAIBChannelSlot	`void sk_setAIBChannelSlot(UBYTE *addrInfo, UBYTE x);`	`void skts_setAIBChannelSlot( UBYTE *addrInfo, UBYTE x);`	0x01 Slot
getAIBSlotB	`XBYTE sk_getAIBSlotB(const UBYTE *AIBBlock);`	`XBYTE skts_getAIBSlotB(const UBYTE *AIBBlock);`	0x01 Slot
setAIBSlotB	`void sk_setAIBSlotB(UBYTE *addrInfo, XBYTE x);`	`void skts_setAIBSlotB(UBYTE *addrInfo, XBYTE x);`	0x01 Slot

SS7 Addressing Functions

Name	Syntax	Threadsafe Syntax	AIB used

getAIBSubrate	`UBYTE sk_getAIBSubrate( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBSubrate( const UBYTE *AIBBlock );`	0x03
setAIBSubrate	`void sk_setAIBSubrate( UBYTE *AIBBlock, UBYTE aSubrate );`	`void skts_setAIBSubrate( UBYTE *AIBBlock, UBYTE aSubrate );`	0x03
setAIBSubrateEntity	`void sk_setAIBSubrateEntity ( UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel, UBYTE aSubrate );`	`void skts_setAIBSubrateEnti ty( UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel, UBYTE aSubrate );`	0x03
getAIBTerminal	`UBYTE sk_getAIBTerminal( const UBYTE *AIBBlock );*/`	`UBYTE skts_getAIBTerminal( const UBYTE *AIBBlock );*/`	0x04
setAIBTerminal	`void sk_setAIBTerminal( UBYTE *AIBBlock, UBYTE aTerminal);`	`void skts_setAIBTerminal( UBYTE *AIBBlock, UBYTE aTerminal);`	0x04
setAIBTerminalEntity	`void sk_setAIBTerminalEntit y(UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel, UBYTE aSubrate, UBYTE aTerminal);`	`void skts_setAIBTerminalEnt ity(UBYTE *AIBBlock, XBYTE aSpan, UBYTE aChannel, UBYTE aSubrate, UBYTE aTerminal);`	0x04
getAIBProfile	`UBYTE sk_getAIBProfile( const UBYTE *AIBBlock);`	`UBYTE skts_getAIBProfile( const UBYTE *AIBBlock);`	0x06 ISDN Call Reference
setAIBProfile	`void sk_setAIBProfile( UBYTE *AIBBlock, UBYTE aProfile );`	`void skts_setAIBProfile( UBYTE *AIBBlock, UBYTE aProfile );`	0x06 ISDN Call Reference
getAIBCallReference	`UBYTE sk_getAIBCallReference ( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBCallReferen ce(const UBYTE *AIBBlock );`	0x06 ISDN Call Reference
setAIBCallReference	`void sk_setAIBCallReference (UBYTE *AIBBlock, UBYTE aCallReference);`	`void skts_setAIBCallReferen ce(UBYTE *AIBBlock, UBYTE aCallReference);`	0x06 ISDN Call Reference
setAIBCallReferenceE ntity	`void sk_setAIBCallReference Entity (UBYTE *AIBBlock, UBYTE aSlot, UBYTE aProfile, UBYTE aSubrate, UBYTE aCallReference);`	`void skts_setAIBCallReferen ceEntity (UBYTE *AIBBlock, UBYTE aSlot, UBYTE aProfile, UBYTE aSubrate, UBYTE aCallReference);`	0x06 ISDN Call Reference

getAIBBaseCICNumber	`int sk_getAIBBaseCICNumber( const UBYTE *AIBBlock );`	`int skts_getAIBBaseCICNumber( const UBYTE *AIBBlock );`	0x14 SS7 CIC Group
setAIBBaseCICNumber	`void sk_setAIBBaseCICNumber (UBYTE *AIBBlock, int aBaseCICNumber);`	`void skts_setAIBBaseCICNumber(UBYTE *AIBBlock,int aBaseCICNumber);`	0x14 SS7 CIC Group
getAIBBaseCICSpan	`XBYTE sk_getAIBBaseCICSpan( const UBYTE *AIBBlock);`	`XBYTE skts_getAIBBaseCICSpan( const UBYTE *AIBBlock);`	0x14 SS7 CIC Group
setAIBBaseCICSpan	`void sk_setAIBBaseCICSpan (UBYTE *AIBBlock, XBYTE aCICSpan);`	`void skts_setAIBBaseCICSpan (UBYTE *AIBBlock, XBYTE aCICSpan);`	0x14 SS7 CIC Group
getAIBBaseCICChannel	`UBYTE sk_getAIBBaseCICChannel(const UBYTE *AIBBlock);`	`UBYTE skts_getAIBBaseCICChannel(const UBYTE *AIBBlock);`	0x14 SS7 CIC Group
setAIBBaseCICChannel	`void sk_setAIBBaseCICChannel(UBYTE *AIBBlock, UBYTE aCICChannel);`	`void skts_setAIBBaseCICChannel(UBYTE *AIBBlock, UBYTE aCICChannel);`	0x14 SS7 CIC Group
getAIBNumCICs	`UBYTE sk_getAIBNumCICs( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBNumCICs( const UBYTE *AIBBlock );`	0x3C Number of CICs
setAIBNumCICs	`void sk_setAIBNumCICs( UBYTE *AIBBlock, UBYTE aNumCICs);`	`void skts_setAIBNumCICs( UBYTE *AIBBlock, UBYTE aNumCICs);`	0x3C Number of CICs
setAIBCICGroupEntity	`void sk_setAIBCICGroupEntity(UBYTE *AIBBlock, UBYTE aBaseCICNumber, UBYTE aBaseCICSpan, UBYTE aBaseCICChannel, UBYTE aNumCICs );`	`void skts_setAIBCICGroupEntity(UBYTE *AIBBlock, UBYTE aBaseCICNumber, UBYTE aBaseCICSpan, UBYTE aBaseCICChannel, UBYTE aNumCICs );`	0x14 SS7 CIC Group
setAIBV5ID	`void sk_setAIBV5IDAndLink( UBYTE *AIBBlock, unsigned short aV5ID, UBYTE aLinkID );`	`void skts_setAIBV5IDAndLink( UBYTE *AIBBlock, unsigned short aV5ID, UBYTE aLinkID );`	0x2D V5 ID, Link ID
setAIBV5IDAndUserPort	`void sk_setAIBV5IDAndUserPort( UBYTE *AIBBlock, unsigned short aV5ID, unsigned short aUserPort);`	`void skts_setAIBV5IDAndUserPort( UBYTE *AIBBlock, unsigned short aV5ID, unsigned short aUserPort);`	0x2F V5 Interface ID, User Port

getAIBStackID	`UBYTE sk_getAIBStackID (const UBYTE *AIBBlock );`	`UBYTE skts_getAIBStackID (const UBYTE *AIBBlock);`	0x08 0x09 0x14
setAIBStackID	`void sk_setAIBStackID( UBYTE *AIBBlock, UBYTE aStackID);`	`void skts_setAIBStackID( UBYTE *AIBBlock, UBYTE aStackID);`	0x08 0x09 0x14
getAIBLinkID	`UBYTE sk_getAIBLinkID( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBLinkID( const UBYTE *AIBBlock );`	0x09 SS7 Link
setAIBLinkID	`void sk_setAIBLinkID( UBYTE *AIBBlock, UBYTE aLinkID);`	`void skts_setAIBLinkID( UBYTE *AIBBlock, UBYTE aLinkID);`	0x09 SS7 Link
setAIBStackEntity	`void sk_setAIBStackEntity( UBYTE *AIBBlock, UBYTE aStackID, UBYTE aLinkID);`	`void skts_setAIBStackEntity (UBYTE *AIBBlock, UBYTE aStackID, UBYTE aLinkID);`	0x09 SS7 Link
getAIBObjectInstanceID	`XBYTE sk_getAIBObjectInstanceID(const UBYTE *AIBBlock);`	`XBYTE skts_getAIBObjectInstanceID(const UBYTE *AIBBlock);`	0x53 Object Type
setAIBObjectInstanceID	`void sk_setAIBObjectInstanceID( UBYTE *AIBBlock, XBYTE anObjectInstanceID);`	`void skts_setAIBObjectInstanceID( UBYTE *AIBBlock, XBYTE anObjectInstanceID);`	0x53 Object Type
getAIBObjectType	`XBYTE sk_getAIBObjectType( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBObjectType( const UBYTE *AIBBlock );`	0x53 Object Type
setAIBObjectType	`void sk_setAIBObjectType( UBYTE *AIBBlock, XBYTE anObjectType);`	`void skts_setAIBObjectType( UBYTE *AIBBlock, XBYTE anObjectType);`	0x53 Object Type
getAIBModule	`UBYTE sk_getAIBModule( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBModule( const UBYTE *AIBBlock );`	0x42 VoIP Module
setAIBModule	`void sk_setAIBModule( UBYTE *AIBBlock, UBYTE aModule);`	`void skts_setAIBModule( UBYTE *AIBBlock, UBYTE aModule);`	0x42 VoIP Module
getAIBAssignSpanSlot	`UBYTE sk_getAIBAssignSpanSlot( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBAssignSpanSlot( const UBYTE *AIBBlock );`	0x11 Logical/ Physical Span

setAIBAssignSpanSlot	`void sk_setAIBAssignSpanSlot(UBYTE *AIBBlock, UBYTE anAssignSpanSlot );`	`void skts_setAIBAssignSpanSlot(UBYTE *AIBBlock, UBYTE anAssignSpanSlot );`	0x11 Logical/ Physical Span
getAIBAssignSpanOffset	`UBYTE sk_getAIBAssignSpanOffset( const UBYTE *AIBBlock );`	`UBYTE skts_getAIBAssignSpanOffset( const UBYTE *AIBBlock );`	0x11 Logical/ Physical Span
setAIBAssignSpanOffset	`void sk_setAIBAssignSpanOffset( UBYTE *AIBBlock, UBYTE anAssignSpanOffset);`	`void skts_setAIBAssignSpanOffset( UBYTE *AIBBlock, UBYTE anAssignSpanOffset);`	0x11 Logical/ Physical Span
getAIBExpandedNode	`XBYTE sk_getAIBExpandedNode( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBExpandedNode( const UBYTE *AIBBlock );`	0x52*
setAIBExpandedNode	`void sk_setAIBExpandedNode( UBYTE *AIBBlock, XBYTE anExpandedNode);`	`void skts_setAIBExpandedNode( UBYTE *AIBBlock, XBYTE anExpandedNode);`	0x52*
getAIBConferenceID	`XBYTE sk_getAIBConferenceID( const UBYTE *AIBBlock );`	`XBYTE skts_getAIBConferenceID( const UBYTE *AIBBlock );`	0x55 Conference ID
setAIBConferenceID	`void sk_setAIBConferenceID( UBYTE *AIBBlock, XBYTE aConferenceID);`	`void skts_setAIBConferenceID( UBYTE *AIBBlock, XBYTE aConferenceID);`	0x55 Conference ID
* IP Signaling Series 3 Card or Expanded Logical Node ID			

10 Redundancy

Purpose This chapter describes the redundancy functions and redundant application functions, available in SwitchKit. This chapter also describes the SK API messages for application redundancy and LLC redundancy. The chapter provides prototype information, possible error return values, and a description of what the function does.

Most functions return an integer value. Upon successful completion, that return value is **OK** (0). Non-zero return values indicate some error conditions. Each function lists the associated error conditions.

Some functions return pointers; those functions do not have a status value indicating success or failure.

sk_activateExnetMatrix()

Description

The ***sk_activateExnetMatrix***() function activates the standby EX/CPU. This message is also available as ***sk_activateExnetMatrixOnConnection()*** and ***skts_activateExnetMatrix()***.

Syntax

```
int sk_activateExnetMatrix(int isLeftFlag, int aNodeID);
int sk_activateExnetMatrixOnConnection(int isLeftFlag,
   int aNodeID, int aConID);
```

Threadsafe Syntax

```
int skts_activateExnetMatrix(int isLeftFlag, int aNodeID,
   int aConID);
```

Parameters

The function parameters are shown in the table below.

Argument	Description
isLeftFlag	isLeftFlag!= 0 indicates that the left EX/CPU is activated. isLeftFlag = 0 indicates that the right EX/CPU is activated. If the EX/CPU to be activated is already active, then no action will occur.
aNodeID	Specifies the node the function is sent to.This argument must be set.
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with

Return Values

Possible return values for this function:

SK_LOST_LLC This return value indicates that your application lost contact with the LLC.

sk_registerAsRedundantApp() / sk_deregisterAsRedundantApp()

Description

Use the function ***sk_registerAsRedundantApp***() when an application wants to register as a member of a RAP. The function then uses the SK_RegisterAsRedundantApp message to process the request. An application can also use this function for parameter changes of a previous registration.

Registering by calling the function instead of directly sending the API message guarantees that the message is resent if the application should lose its connection with the LLC.

This message is also available as ***sk_registerAsRedundantAppOnConnection()*** and ***skts_registerAsRedundantApp()***.

To de-register, the application must issue ***sk_deregisterAsRedundantApp***(). This function also uses the *SK_RegisterAsRedundantApp* message for the de-registration, but the RedundantAppPriority argument is hard coded to -1 in order to remove the application from the RAP. This message is also available as ***sk_deregisterAsRedundantAppOnConnection()*** and ***skts_deregisterAsRedundantApp()***.

The application must be able to handle the acknowledgments for a registration and de-registration.

Syntax

```
int sk_registerAsRedundantApp(int anAppID, const char
   *aRedundantAppPoolID, int aRedundantAppPriority, int
   aDataSize, const UBYTE *aDataBuf, void *aTag,
   HandlerFunc *aHandlerFunc);
int sk_deregisterAsRedundantApp(int anAppID, const char
   *aRedundantAppPoolID, void *aTag, HandlerFunc
   *aHandlerFunc);
int sk_registerAsRedundantAppOnConnection(int anAppID,
   const char *aRedundantAppPoolID, int
   aRedundantAppPriority, int aDataSize, const UBYTE
   *aDataBuf, void *aTag, HandlerFunc *aHandlerFunc, int
   conID);
int sk_deregisterAsRedundantAppOnConnection(int anAppID,
   const char *aRedundantAppPoolID,void *aTag,
   HandlerFunc *aHandlerFunc, int conID);
```

Threadsafe Syntax

```
int skts_registerAsRedundantApp( int anAppID, const char
   *aRedundantAppPoolID, int aRedundantAppPriority, int
   aDataSize, const UBYTE *aDataBuf, void *aTag,
   HandlerFunc *aHandlerFunc, int aConID );
```

```
int skts_deregisterAsRedundantApp( int anAppID, const char
   *aRedundantAppPoolID, void *aTag, HandlerFunc
   *aHandlerFunc, int aConID );
```

Parameters The function parameters are shown in the table below.

Parameter	Description
anAppID	The application identifier assigned by the LLC to the application. The name can be obtained by calling **sk_getConnectionName**().
aRedundantAppPoolID	A string that uniquely identifies the class of application wishing to be treated as redundant applications. ***Note:*** The ID is case sensitive.
aRedundantAppPriority	RedundantAppPriority is a value indicating the current priority of the application. The higher the value, the more likely it is that the LLC will select the application as primary. Special values are: SK_RED_APP_PRI_MONITOR (0) Used by an application wishing to monitor the activity of this RedundantAppClass. An application selecting this priority cannot be considered either primary or secondary. SK_RED_APP_PRI_REMOVED (-1) Used when an application needs to be removed from an RedundantAppClass for which it has previously registered.
aDataSize	DataSize
aDataBuf	aData can be used by the application to store information useful to this application. The value specified in the request is returned when the RAP is queried via the *SK_RedundantAppQuery* message.
aTag	An application defined pointer which will be returned to the application when a handler is invoked.
aHandlerFunc	A pointer to a function that is designed to handle messages.

Parameter	Description
aConID	aConID is a connection identifier specified at connection creation time and used to indicate which LLC an application wishes to communicate with

LLC and Application Redundancy

Description

Applications developed using SwitchKit must often be deployed in a redundant fashion, to allow for application and host computer failures. You can accomplish application redundancy by starting multiple, identical copies of an application, preferably on seperate host computers. Each of these applications must be capable of independently carrying out the other applications functions.

The LLC supports redundant applications by allowing distinct processes to register as redundant applications. This construct is named the Redundant Application Pool (RAP). The LLC assists redundant applications by designating one process to be the primary application for a particular RAP.

The LLC monitors all applications that are members of a RAP, so that one and only one of these applications is the primary application. If the primary application should fail, the LLC designates another of these applications to be the primary application.

Rules of RAP

The following table shows the general rules of SwitchKit Application Redundancy.

Rule	Definition
1	The first member of a RAP is primary. All subsequent members are secondary.
2	If the primary application fails or terminates, the application loses its primary status.
3	A member of a RAP becomes primary based upon the priority settings specified at registration time.

Tasks of the LLC

The following list explains the tasks of the LLC regarding redundant application settings. The LLC

- maintains RAPs.
- maintains at most one primary app within each RAP.
- adds applications to a RAP as specified in the ***sk_registerAsRedundantApp***() call from the application.
- determines which application is primary in the RAP.

- notifies the application of its redundancy status upon entering the RAP.
- notifies all members of a RAP about any status change of member applications.
- removes an application from all RAPs if disconnected or not responding.
- allows an application to force itself to be primary in a RAP.
- allows an application to remove itself from a RAP.

Tasks of the Redundant Application

The following list explains the responsibility of applications that are members of a RAP.

- Upon indication that an application is the primary application, it must take control of all resources belonging to the application.
- Upon indication that an application is a secondary application, it must relinquish control of all resources belonging to the application.
- The primary and secondary applications are responsible for sharing information among themselves, so that the state of the application is maintained and known across all members of the RAP. This means that if the primary application of a RAP receives resources (such as a channel or active call) it should notify all secondary applications. This is necessary because if something should happen to the primary application, the secondary application that takes over must have sufficient information to continue processing all active calls, without any loss of service or resources.

RAPs and Redundant LLCs

In a system setup with redundant LLCs, the redundant RAP information is not shared between the active and standby LLCs. All registration messages and calls are saved within the SwitchKit API in the application's process space. When the connection to the active LLC is lost, the SwitchKit API attempts to connect to the redundant LLC. Upon successful connection, the SwitchKit API resends any previously sent redundant application registration messages. This approach keeps the time where no primary application is selected to a minimum.

When an LLC switchover occurs, it is possible that the primary application can become secondary, based on the order of the applications re-registration as initiated by the SwitchKit API. If this occurs, the previously secondary application will also be told that it is now primary.

If the application wishes to correct this situation, the new primary application must send a ***ReselectPrimaryApp*** message to the LLC indicating that it wishes to be made secondary. The LLC then selects another primary application from the registered applications (if another application exists).

Response Values

The following table shows the Status field values for the redundant application support:

If the return value is	then...
SK_RED_APP_POOL_DOESNT_EXIST (0Xf00c)	the queried RAP does not exist.
SK_RED_APP_MEM_ERROR (0xf00d)	an error occured under the members of a RAP.
SK_RED_APP_NOT_A_MEMBER (0xf00e)	the application is not a registered member of the RAP.
SK_RED_APP_NOT_CONNECTED (0xf00f)	the registered application lost its connection to the LLC.

API Messages used for Redundancy

API Messages

The API messages listed below are used for redundancy. Information about these API messages can be found in the *API Reference*.

- Reduntant App Pool Member Query
- RedundantAppQuery
- RedundantAppQuery
- RedundantAppStatusMsg
- Redundant LLC Register
- RegisterAsRedundantApp
- Reselect Primary App

11 Programming Tips & Examples

Purpose This chapter provides information on parsing address information blocks and advanced programming technique in UNIX.This chapter also explains calls into the SwitchKit API which are necessary for a call processing application. The examples provide a template for building your own call processing application.

> **Important!** The object of the section showing examples is not to provide a complete call processing solution.

The actual code for the sample applications can be found in the installation directory in *samples* and the configuration file can be found in *samples/cfg*. The demonstration contains two processes, SimpleTandem and CallSim that run in conjunction with each other, whereby SimpleTandem answers calls that have been initiated by CallSim.

Parsing of AIBs

Description

Upon receiving a message that contains an AIB, such as *XL_DS0StatusChange*, it is possible to get the Span and Channel by using the **sk_getAIB*** routines.

Code Example

```
int genericHandler(SK_Event *evt, void *tag) {
   MsgStruct *msg = evt->IncomingMsg;

   SK_MSG_SWITCH(msg) {
      /* Print info about any DS0 status change */
      CASE_DS0StatusChange(ds0) {
         printf("DS0 Stat Change, Span:%d, Chan:%d,
                 Stat:%d, Purge:%X\n",
                 sk_getAIBSpan(ds0->AddrInfo),
                 sk_getAIBChannel(ds0->AddrInfo),
                 ds0->ChannelStatus,
                 ds0->PurgeStatus);
         return OK;
      }

      CASE_default {
         printf("Unknown message\n");
         return OK;
      }
   } SK_END_SWITCH;
}
```

Advanced Programming Technique in UNIX

Description

In UNIX, an application can split into two or more processes after opening a connection with the LLC. The processes, parent and child have a copy of the connection. All processes can write to the connection with no problem, but if they all try to read, a race condition occurs and only one process receives the message.

How to Solve the Race Condition

In case parent and child need to communicate with the LLC, the child should call **sk_initializeConnection**() or one of the **sk_createConnection**() functions after the split to get its own connection.

It is important to understand that in such a case, the processes have a completely different connection. An acknowledgment of a message sent by one process cannot be received by the other process. Use the message *SK_InterAppMsg* to pass on acknowledgments or other important messages.

Channels are also associated with connection names. Even after a split of processes, the channels stay with the parent. To delegate a channel assignment to a child, use the message *SK_TransferChanMsg*.

Example

The following example demonstrates the connection management calls in action. In this example, a UNIX process forks a child that connects to the LLC socket.

```
sk_broadcastLoad(1);// This makes the (parent) open its
   socket

child = fork();   // Spawn a child process
 if (child==0)
   {              // CHILD PROCESS CODE
      SK_Connection *parentConnect;
      SK_Connection *newConnect;
                  // save the parent's socket
      parentConnect = sk_getCurrentConnection();
      newConnect = sk_createConnection("Child",-1,0,NULL,
                     -1,NULL,-1);
                // set the child's socket to the new one
      sk_setCurrentConnection(newConnect);
                // close the parent's socket in the child
      sk_destroyConnection(parentConnect);
```

```
};
```

List of Related Connection Management Functions

Depending on your application, you have to issue a function call to one or more of the following functions after splitting a process:

- SK_Connction ***sk_createConnection***();
- SK_Connction ***sk_createConnectionWithID***();
- **sk_*OnConnection**();
- **sk_initializeConnection**();
- **sk_initializeForcedConnection**();

List of Related API Messages

Depending on your application, you need to send one or more of the following API messages:

- *SK_InterAppMsg*
- *SK_TransferChanMsg*

Simple Tandem and CallSim

Description

The following information provides insight into a simple tandem call control application included with the samples on the EXS SwitchKit installation CD.

Simple Tandem

The SimpleTandem application waits after initialization for a Request For Service With Data (RFSD) message. When an RFSD comes in, it outseizes and connects two channels based on a simple, hard-coded routing algorithm. It performs this function each time a new RFSD is received. It can be used in conjunction with CallSim.

To run SimpleTandem, the switch must be configured with the provided tandem.cfg configuration file.

Please refer to the comments in SimpleTandem.h and SimpleTandem.cpp. These files are provided in PDF format, and you can use the established links to access them.

Process Overview

- Initialize a connection to the LLC and inform it of the groups being watched by the application.
- Wait for RFSD in <orig> group.
- Call the Route Class to get the number from RFSD and check the AreaCode Class to determine the <dest> group to be used.
- Create an Outseize Control message and outseize to <dest>.
- Upon successful outseizure, connect the inbound and outbound channels.
- Upon release of the inbound or outbound channel, release the connected channel and return both channels to the channel manager.

Call Simulator

The CallSim application simulates the endpoints of a tandem call and manages two channel groups, the originating channel group and the destination channel group.

Using the SwitchKit UserTimer message, a timer is set up to outseize a channel from the <orig> group on a regular interval. You can define this interval by passing in the CallDelay argument. Upon succesful outseizure, this channel is parked for a predetermined length of time.

You can define the length of time by passing in the CallDuration argument. After CallDuration number of seconds, the channel is released.

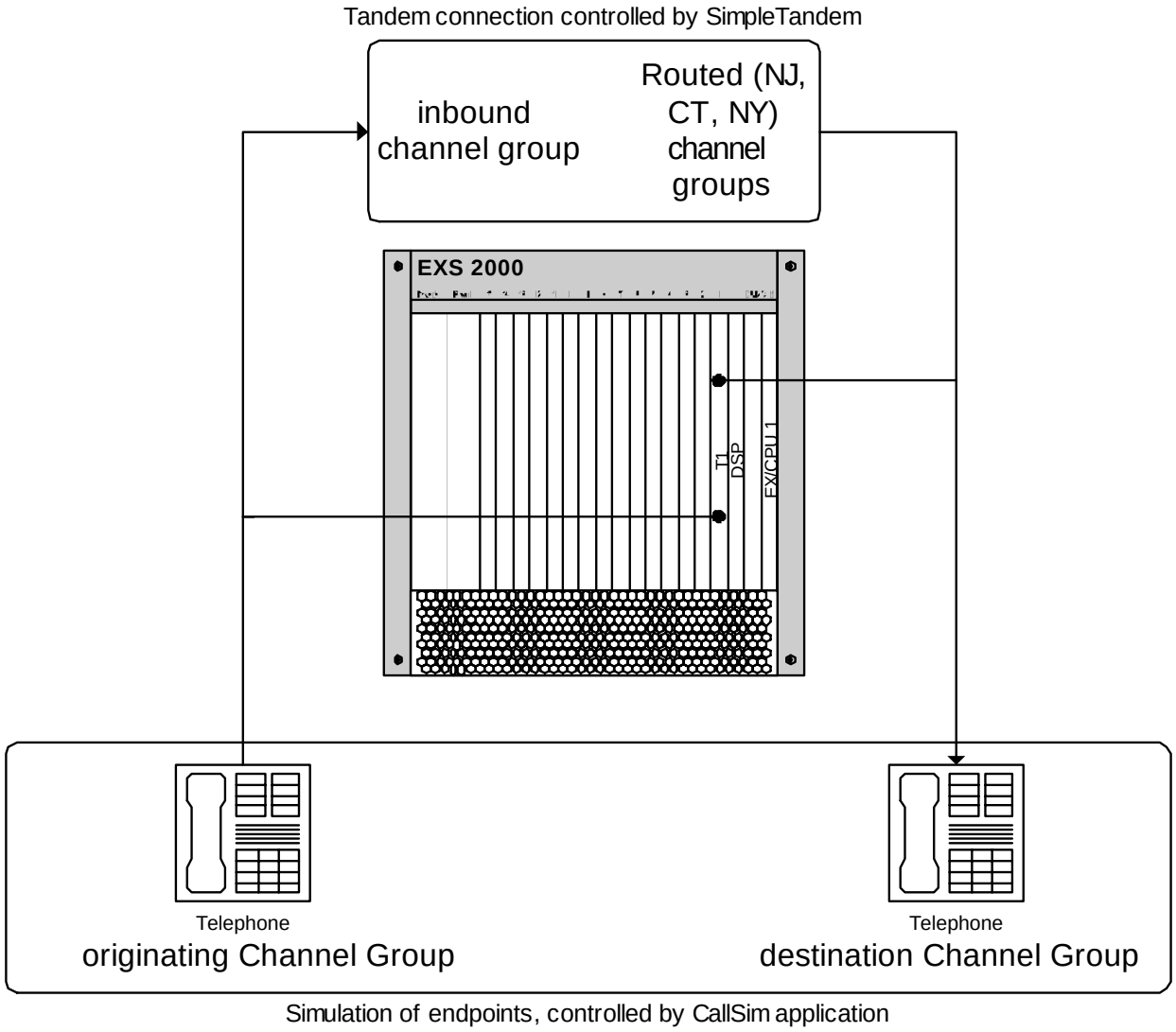
To run CallSim, the switch must be configured with the provided tandem.cfg configuration file.

Process Overview

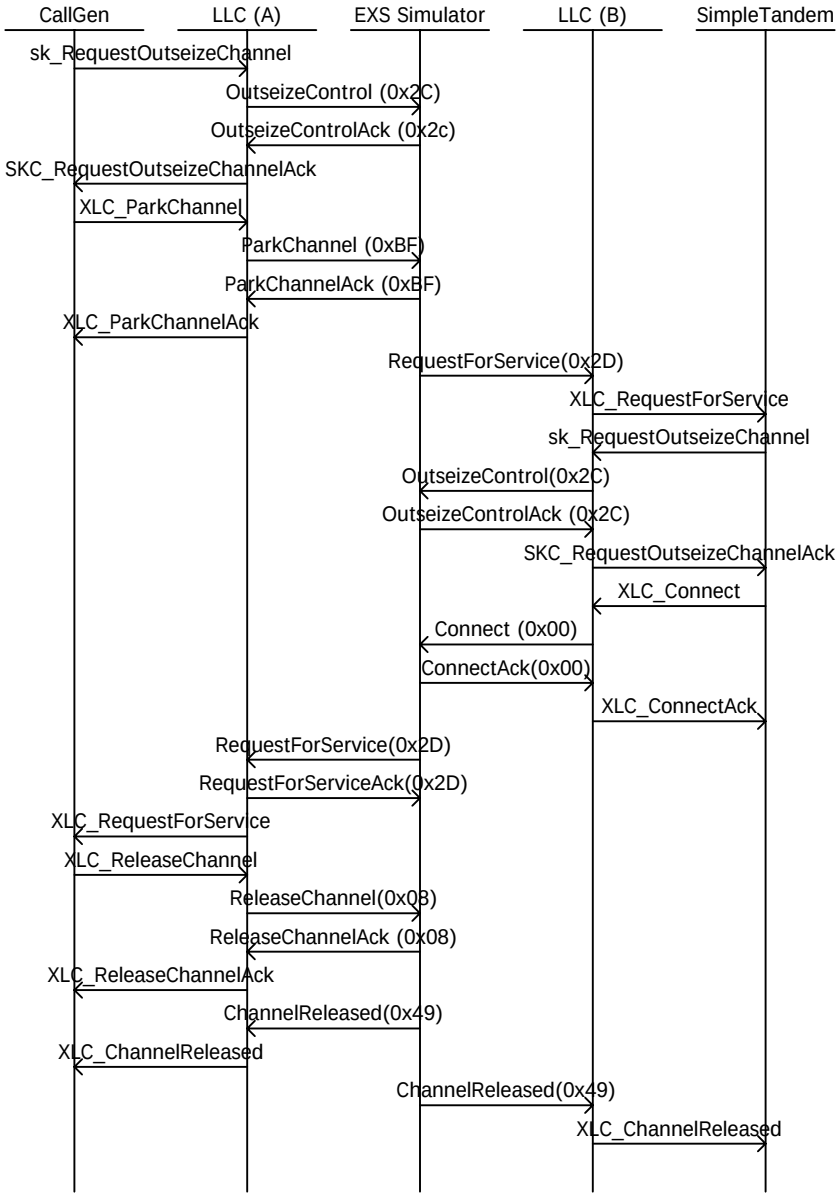
- Initialize a Connection with the LLC.
- Set the group handler.
- Set up a UserTimer message to trigger a certain number of outseizures. This is set by the NumberOfCalls parameter every CallDelay seconds.
- Construct the Outseize Control message and outseize.
- On receiving positive acknowledgement, park the channel for the CallDuration period of time, and then release the channel.

- When the application receives a 'ChannelReleased' message (at genericHandler), the channel is returned.

Illustration Diagram



Call Flow


CallGen
LLC (A)
EXS Simulator
LLC (B)
SimpleTandem
sk_RequestOutseizeChannel
OutseizeControl (0x2C)
OutseizeControlAck (0x2c)
SKC_RequestOutseizeChannelAck
XLC_ParkChannel
ParkChannel (0xBF)
ParkChannelAck (0xBF)
XLC_ParkChannelAck
RequestForService(0x2D)
XLC_RequestForService
sk_RequestOutseizeChannel
OutseizeControl(0x2C)
OutseizeControlAck (0x2C)
SKC_RequestOutseizeChannelAck
XLC_Connect
Connect (0x00)
ConnectAck(0x00)
XLC_ConnectAck
RequestForService(0x2D)
RequestForServiceAck(0x2D)
XLC_RequestForService
XLC_ReleaseChannel
ReleaseChannel(0x08)
ReleaseChannelAck (0x08)
XLC_ReleaseChannelAck
ChannelReleased(0x49)
XLC_ChannelReleased
ChannelReleased(0x49)
XLC_ChannelReleased

Building the Application

Description All path names are relative to the directory entered during the SwitchKit installation process.

UNIX The Makefile included in the samples/src directory assumes the use of GNU make 3.74 or higher. It automatically compiles and links CallSim and simpleTandem as well as all of the other sample applications in the samples directory. The Makefile automatically determines which of the recommended platforms and compilers is being used. Executing the make command automatically builds all samples.

UNIX Troubleshooting

- Syntax errors in Makefile on any UNIX platform:
  - Make sure that the make utility you are using is at least GNU make 3.74 or higher. If you don't have access to this utility, then remove all platform definitions that are not relevant for the platform you wish to develop on.
- Compilation or link errors:
  - Make sure that the paths to lib and include directories are correct.

Building with Linux Red Hat 6.0 - 6.2

1. Install all compat-egcs*.rpm files from the distribution disk (/mnt/cdrom/RedHat/RPMS)
   - rpm -Uvh compat-egcs*.rpm

   Important! You must have root access to complete this step!

2. cd to /samples/src and edit the Makefile. The following code appear near line 22:

```
ifeq ($(shell uname -s),Linux)
  CCC = g++
  CC = gcc
  EXTRALIBS =-Ipthread
endif
```

   Change the path name for the location of the 5.2 compatible compiler tools so that it is in CCC.

3. Make sure that you are not the root user when you start the build.

Windows® NT The directory, samples/NTBuild contains a Visual C++® version 5.0 workspace, named samples.dsw. This workspace contains projects for each sample application included with the SwitchKit installation. After opening the workspace, right-click the project you want to build in the File View tab, then choose Build (selection only). The executables

appears in samples/NTBuild/Debug/SimpleTandem or samples/NTBuild/Debug/CallSim. If the release version of the applications are built, the executables appears in samples/NTBuild/Release/SimpleTandem or samples/NTBuild/Release/CallSim.

Windows® Troubleshooting

Follow these instruction for problems, if you attempt to start a new project or if you want to change the workspace.

- If you get the following error message:

```
Compilation error: fatal error C1083: Cannot open
include file: 'SKC_API.h': No such file or directory
```

  - Go to **Project Settings**,
  - Select the **C/C++** tab,
  - Under **Category**, select Preprocessor,
  - Make sure that the field “Additional include directories” shows:
    ../../include

- If you get the following error message:

```
Link Error: CallSim.obj : error LNK2001: unresolved
external symbol "public: virtual __thiscall
SKC_UserTimer::~SKC_UserTimer(void)"
(??1SKC_UserTimer@@UAE@XZ)
```

  - Make sure that include/BaseClasses.cpp and include/Messages.api.cpp are members of the project.

- If a run time error occurs:
  - Go to **Project Settings**,
  - Select the **C/C++** tab,
  - Under Category, select Code Generation
  - Set the field “Use run-time library” to Multithreaded DLL.

Linking to the SwitchKit API Library file

When you write an application you must link your application to the SwitchKit API library file: *skapi.lib*.

To link to the library file, for example, using Microsoft® Visual C++® Version 6.0, you would do the following steps.

1. On the menu select **Project→Settings**. The **Project Settings** dialog box opens.
2. Select the **Link** tab.
3. Type after the existing text in the text box for **Object/library modules:** `skapi.lib`.

4. Add the path to the library file in the text box for **Project Options:**

```
/libpath:"C:\Program Files\ Cantata\Switchkit\lib"
```

5. Click **OK**.

Proper Initialization

An error message will result, as described further on, when a message is sent before being properly initialized. You must call sk_initMsg() before sending the message. If the application should fail to call sk_initMsg() before sending the message, the following text will appear on the console and in the application's maintenance log:

```
   "**Error:Aug 12 2003 09:30
:30: Received message with unknown engine type205 - is
   the sender of this message incorrectly using a pre-3.0
   version of SwitchKit? This version of SwitchKit will
   not work with any pre-3.0versions."
```

Running the Application

Description

You can run your call control application described below.

Running the Applications

To run the applications, perform the following steps:

- Start the LLC.
- Start SwitchManager and send the tandem.cfg file with the defined inbound and outbound channel groups.
- Start SimpleTandem at the command line:
  - SimpleTandem
- Start CallSim at the command line:
  - CallSim

To see usage information for either of the applications, provide an -h argument at the command line.

12 Threadsafe SwitchKit API

Purpose This chapter provides an explanation of the Threadsafe SwitchKit API library for application development.

Threadsafe Introduction

Overview

The SwitchKit Threadsafe library is an optional replacement for the standard library provided with the SwitchKit development package. It contains the standard SK API for non multi-threaded applications as well as threadsafe versions of each C function and C++ method as necessary. These new functions and methods are the only functions and methods that should be used by the application developer wishing to use the SwitchKit threadsafe library.

> **Important!** The SwitchKit Threadsafe library for Linux is a shared not a static library.

Notational Conventions

All new functions in the threadsafe library begin with the ***skts_*** prefix instead of the customary ***sk_*** of the standard SwitchKit library. There is one additional method in the SKC_Message class, *tsSend()*, that all message classes inherit from, which allows messages to be sent safely in a multi-threaded environment. Failure to use the appropriate functions once thread safety has been enabled will result in the following error being logged in the application's maintenance log:

Where ***someFunctionName()*** is the actual function that was improperly called:

```
"Function call made to unsafe API(someFunctionName())
   when Thread Safety enabled."
```

CAUTION

Applications should not mix threadsafe function calls with non-threadsafe SwitchKit function calls. Failure to heed this warning will cause unexpected and potentially hazardous results.

Sample Applications

Purpose

This section presents a comparison of a sample threadsafe application and a sample of non-threadsafe application, both written in a SwitchKit API development environment.

Sample Non-Threadsafe Application

The following shows a simple application both using the standard SwitchKit library.

What the application does

1. Connects with an LLC
2. Sets up handlers:

   A default handler that will receive all messages should other handlers not be enabled to catch a message

   A group handler for receiving all inbound call notifications via the RequestForService or RFSWithData messages
3. Registers for PPLEventIndication and DS0StatusChange messages
4. Loops forever waiting for a message from the LLC.

Sample Non-Threadsafe Code

```
main(int argc, int argv)
{
   ...
   // (1) Establish connections to LLC's
   int status = skts_createConnection sk_ method(
                1,    // connection ID
                -1,   // application ID (allow LLC to select one)
                0,    //isForced (set to 0)
                "host1", // primary LLC host name
                1312, // primary LLC port number (1312 is default)
                "host2", // redundant LLC hostname
                1312); // redundant LLC port number 1312
   if (status == NULL)
   {
        cout << ìError has occurred creating connection to LLCî << endl;
        return (-1);
   }
   ...
   // (2) Define Handler Functions - returns OK always
   // (2a) Default Handler
   (void)skts_setDefaultHandler(someTag,                // tag for hdlr
                                someDefaultHandlerFunc); // handler func

   // (2b) Set Group Handler for inbound calls
   (void)sk_setGroupHandler("inboundChannelGroup",// group name
```

```
                            NULL,   // tag for hdlr
                            someGroupHandlerFunc);// handler func
   ...
   // (3) Register for unsolicited messages
   status = sk_msgRegisterOnConnection(
             SK_PPLEIS | SK_DSOS,  // registration mask
             1);                   //connection ID
if (status != OK) {
     cout << "Error has occurred registering for message with LLC"
          << endl;
     return (-1);
   }
(4) Register for new calls from the specified channel group
status = sk_watchChannelGroupOnConnection(
        "inboundChannelGroup", // group name
        1);                    //connection ID
if status !=OK {
     cout << Error requesting watch of channel group"
          << endl;
     return (-1);
}
...
   ...
// (5) SwitchKit receives loop
while(1) {
      char *buffer;  //where msg will be returned if not  handled
      int size;      // size of msg returned
      void *data;    //needed for function call. Not used by API

      status = sk_rcvAndDispatchAutoStorage(
                  &buffer,   // contains packed msg if msg not handled
                  &size,    // size of message returned
                  -1,       // timeout - wait forever
                  &data);  // Not currently used.

     if (status != OK) {
            switch(status) {
                case SK_NO_MESSAGE:
                  // This is a normal return and indicates that
                  // an internal message was returned.
                  break;

                case SK_BAD_MESSAGE:
                    cout << " Bad Message rcvd from SwitchKit "
                         << endl;
                    break;

               case SK_LOST_LLC:
                    cout << " Lost Contact With LLC "
                    << endl;
```

```
               break;

case SK_NOT_HANDLED:
            {
            // Message was not processed by any
            // application defined handlers
            SKC_Message *inboundMsg;
            int retval = skc_unpackMessage(
                              buffer,
                              size,
                              &inboundMsg);
            if (retval != OK)
            {
                 cout << "Error unpacking message"
                      << endl;
                 break;
            }
            cout << "Received message "
                 << inboundMsg->getMsgName()
                 << endl;
            break;
}
        } // end switch
     } // end if
} // end while
...
}
```

Sample ThreadSafe Application

In order to convert the simple application presented above into a threadsafe application, you must make the following changes (the text in the right column show how to update the previous application's code to make the application threadsafe.

What the non- threadsafe application does...	What to do to make the previous application threadsafe....
(0) Not applicable	The **skts_enableThreadSafeLib()** function initializes some global process level data and enables detection of the mixing of threadsafe and non threadsafe functions. This function must be called first in a multi-threaded environment before any other SwitchKit calls. Failure to call this function first will result in unexpected behavior.
(1) Connects with an LLC	Convert **sk_createConnectionWithID()** to the threadsafe version of the function, **skts_createConnection()**. The arguments to both functions are the same. The only difference is the **sk_createConnectionWithID()** returns **SK_Connection *** while the threadsafe version returns a connection ID (aConID), if the connection is successful or previously created. If **skts_createConnection()** fails the following error is returned: SK_ERROR_CREATING_CONNECTION with a negative integer value. Your application may currently use one of several other ways to establish a connection including **sk_initializeConnection()**, **sk_initializeConnectionForced()** and **sk_createConnection()**. Or, it may not even explicitly connect to the LLC (in which case, the environment variables are used to determine the location of primary and redundant LLC). These functions are not supported for threadsafe applications. The only way to connect to the LLC for threadsafe applications is using **skts_createConnection()**. The connection ID concept allows an application to connect to multiple redundant LLC pairs for scalability. Specifying a connection ID dictates which LLC the SK API will use as the target of a function or message. All threadsafe functions that result in messages being sent to an LLC require a connection ID as the last argument.
(2) Sets up handlers: (a) A default handler that will receive all messages should other handlers not be enabled to catch a message (b) A group handler for receiving all inbound call notifications via the RequestForService or RFSWithData messages	(a) Convert **sk_setDefaultHandler()** to the threadsafe version of the function, **skts_setDefaultHandler()**. The arguments to both functions are the same. (b) Convert **sk_setGroupHandler()** to the threadsafe version of the function, **skts_setGroupHandler()**.The arguments to both functions are the same.

What the non- threadsafe application does...	What to do to make the previous application threadsafe....
(3) Registers for PPLEventIndication and DS0StatusChange messages	Convert **sk_msgRegisterOnConnection()** to the threadsafe version of the function, **skts_msgRegister()**. The arguments to both functions are the same. Your application may use a slightly different registration function **sk_msgRegister()**. As mentioned in (1) above, all threadsafe functions require the use of a connection ID to indicate which LLC to target. Note: If your application previously used **sk_msgRegister()**, use the same connection ID value specified in a previous **skts_createConnection()** function.
(4) Registers with the LLC for new calls arriving on channels within the specified group.	Convert **sk_watchChannelGroupOnConnection()** to the threadsafe version of the function, **skts_watchChannelGroup()**. The arguments are identical. Your application may use a slightly different watch channel group function, **sk_watchChannelGroup()**.As mentioned in (1) above, all threadsafe functions require the use of a connection ID to indicate which LLC to target. **Note:** If your application previously used **sk_watchChannelGroup()**, use the same connection ID value specified in a previous **skts_createConnection()** function.
(5) Loops forever waiting for a message from the LLC.	Convert **sk_rcvAndDispatchAutoStorage()** to the threadsafe version of the function, **skts_rcvAndDispatchAutoStorage()**. The arguments are nearly identical with the exception of the last argument, a connection ID. For **skts_rcvAndDispatchAutoStorage()** and **skts_rcvAndDispatch()**, the connection ID argument is an output argument and is used to indicate which LLC connection generated the message should the message be returned in **skts_rcvAndDispatch()/ skts_rcvAndDispatchAutoStorage()**. This would only happen if **skts_rcvAndDispatch()/ skts_rcvAndDispatchAutoStorage()** returns SK_NOT_HANDLED.

Sample Threadsafe Code

The modified code is shown below. Any changes that are necessary are noted in bold text.

```
main(int argc, int argv)
{
     ...
     // (0) Initialize the threadsafe library.
     // This function will initialize some global process level data and
     // enable the detection of the mixing of Threadsafe with non Threadsafe     //
    //functions. This function MUST be called first. Failure to call
     // this function first or at all will result in unexpected behavior.
     skts_enableThreadSafeLib() ;

     // (1) Establish connections to LLC's
     int status = skts_createConnection(
```

```
                                        1,    // connection ID
                                        example, // a const char * to label the
                              connection
                                        -1,   // application ID (allow LLC to
                              select one)
                                        0,    // isForcedFlag (set to 0)
                                        "host1",// primary LLC host name
                                        1312,// primary LLC port number (1312 is
                              default)
                                        "host2",// redundant LLC hostname
                                        1312);// redundant LLC port number
                              (1312)

    if (status <0)
    {
         cout << "Error has occurred creating connection to LLC" <<
  sk_statusText(status)<<"("<<status<<")"<<end1;
         return (-1);
  }
  ...
  // (2) Define Handler Functions - returns OK always
  // (2a) Default Handler
  (void)skts_setDefaultHandler(someTag,// tag for hdlr
                           someDefaultHandlerFunc);

  // (2b) Group Handler to register for inbound calls
  (void)skts_setGroupHandler("inboundChannelGroup",// group name
                           NULL,     // tag for hdlr
                           someGroupHandlerFunc);// handler func
  ...
  // (3) Register for unsolicited messages
  status = skts_msgRegister(
         SK_PPLEIS | SK_DSOS, //registration mask
         1);                  // connection ID
  if (status != OK) {
         cout << "Error has occurred registering for message with LLC"
              << endl;
         return (-1);
  }
  ...
  // (4) Register for new calls from a specified group
  status = skts_watchChannelGroup (
    "inboundChannelGroup",// group name
    1);       // connection ID
  if (status != OK) {
       cout << "Error requesting watch of channel group"
            << endl;
       return (-1);

(5) SwitchKit receive loop
while(1) {
      char *buffer;  // where msg will be return if not handled
      int size;      // size of msg returned
```

```
      void *data;   // needed for function call. Not used by API.
      int retConID;   // conn ID of LLC the message was received from

      status = skts_rcvAndDispatchAutoStorage(
               &buffer,// contains packed msg if msg not handled
               &size, // size of message returned
               -1,  // timeout - wait forever
               &data,// Not currently used.
               &retConID);// connection ID on which msg was rcvd

     if (status != OK) {
                 // The code here is exactly the same as in
                 // Sample Non-Threadsafe Code and is excluded for brevity
     } // end if
} // end while
...
}
```

Other Changes Developers Must Make When Using Threadsafe SK API

Message Registration

In the standard SwitchKit library, several functions that resulted in messages being sent to the LLC had to be issued only once during the life of the application. If the LLC were to be disconnected from the application or the LLC were restarted, the SwitchKit API took care of resending the messages to the LLC. The following functions are automatically resent (effectively, the SwitchKit API calls the function internally for the application) whenever a connection to an LLC is re-established:

- sk_appGroupRegister()
- sk_watchChannelGroup()
- sk_ignoreChannelGroup()
- sk_clearChannelGroup()
- sk_msgRegister()
- sk_pplComponentRegister()
- sk_pplTCAPRegister()
- sk_monitorChannel()
- sk_unmonitorChannel()
- sk_broadcastLoad()
- sk_registerAsRedundantApp()
- sk_deregisterAsRedundantApp()

Automatic Resending of Messages Disabled

When using the optional Threadsafe SwitchKit library, the automatic resending of messages feature is disabled. Instead, an application should register an LLC connection handler using the skts_setLLCConnectionHandler(). This connection handler is called any time the connection to the LLC is lost or re-established. When the handler is called, there is an indication of why the handler is called (SK_LLC_CONN_CREATED or SK_LLC_CONN_DESTROYED). When the indication is SK_LLC_CONN_CREATED, all initialization functions should be called, with the exception of skts_enableThreadSafeLib(), skts_setLLCConnectionHandler(), skts_createConnection() and any of the functions which establish handlers. A modified version of threadsafe example program (Modified Threadsafe Sample Code) is presented below. You will notice the following changes:

- Writing of a new handler function (llcConnectionHandler()) that is called whenever the connection state of an application to an LLC changes.
- The relocation of calls to registration functions, skts_msgRegister() (previously identified as step 3) and skts_watchChannelGroup() (previously identified as step 4) from the main() function to the LLC connection handler. This way, the functions will be called each time the application establishes a connection with the LLC.
- In the main() function, code was added to define the LLC connection handler using the skts_setLLCConnectionHandler() function.

All other registrations for messages should take place in the LLC connection handler including registration for PPL component specific notification and registration for TCAP messages.

Modified Threadsafe Sample Code

The sample code follows. The changes are highlighted in bold.

```
// This is the LLC Connection handler which will be called any
// time the state of a connection to an LLC changes.
void llcConnectionHandler(int aConID, // conn ID of LLC
                          int aConState, // new connection state for LLC
                          void *aTag) {// tag specified when hdlr defined
    if (aConState == SK_LLC_CONN_CREATED) {
        // This handler will be called with this state
        // any time a connection is established to an LLC.
        // This includes the first time we establish a
        // connection to the LLC, LLC switchover, recovery
        // from temporary loss of connection, etc.

        ...

        // (3) Register for unsolicited messages
        status = skts_msgRegister(
                SK_PPLEIS | SK_DS0S, // registration mask
                aConID); // connection ID
        if (status != OK) {
                cout << "Error registering for messages from LLC"
                        << endl;
        // Continue to initialize. No point returning.
    }

    // (4) Register for new calls from a the specified
```

```
        // channel group
        status = skts_watchChannelGroupOnConnection(
                    "inboundChannelGroup", // group name
                    aConID);                    // connection ID
}
        if (status != OK) {
            cout << "Error registering for messages from LLC"
    << endl;
            // Continue to initialize. No point returning.
        }
...
    }
    else if (aConState == SK_LLC_CONN_DESTROYED) {
            // This handler will be called with this state
            // any time a connection to an LLC is lost.
            // This includes the temporary loss which occurs
            // as part of LLC switchover in a redundant system

            // Here you may want disable the sending of messages
            // to the LLC the application just disconnected from.
            // all attempts to send will fail until the connection
            // is reestablished.
            ...
    }

main(int argc, int argv) {
   ...
   // (0) Initialize the threadsafe library.
   // This function will initialize some global process level data and
   // enable the detection of the mixing of Threadsafe with non Thread-
   // safe functions. This function MUST be called first. Failure to call
   // this function first or at all will result in unexpected behavior.
   skts_enableThreadSafeAPI();

   // (0.5) Set up the LLC connection handler
   (void)sk_setLLCConnectionHandler(
           NULL,// a user-defined value passed to hdlr func.
           llcConnectionHandler);// LLC connection hdlr func.

   // (1) Establish connections to LLC's
   int status = skts_createConnection(
                1,   // connection ID

                -1,  // application ID (allow LLC to select one)
                0,   // isForcedFlag (set to 0)
                "host1",// primary LLC host name
                1312,// primary LLC port number (1312 is default)
```

```
                "host2",// redundant LLC hostname
                1312);// redundant LLC port number (1312 is default)
   if (status != 1)
   {
     cout << "Error has occurred creating connection to LLC" << endl;
     return (-1);
   }
   ...
   // (2) Define Handler Functions - returns OK always
   // (2a) Default Handler
   (void)skts_setDefaultHandler(someTag, // tag for hdlr
                             someDefaultHandlerFunc);// handler func

   // (2b) Group Handler to register for inbound calls
   (void)skts_setGroupHandler("inboundChannelGroup",// group name
                              NULL,     // tag
                              someGroupHandlerFunc);// handler func
   ...

...
   // (5) SwitchKit receive loop
   while(1) {
       char *buffer;// where msg will be return if not handled
       int size;// size of msg returned
       void *data;// needed for function call. Not used by API.
       int retConID;// conn ID of LLC the message was received from

       status = skts_rcvAndDispatchAutoStorage(
              &buffer,// contains packed msg if msg not handled
              &size, // size of message returned
              -1,  // timeout - wait forever
              &data,// Not currently used.
              &retConID);// connection ID on which msg was rcvd

     if (status != OK) {
          // The code here is exactly the same as in
          // Sample Code 1 and is excluded for brevity.
          ...
     } // end if
} // end while
...
}
```

Linking a Multi-threaded Application

The ThreadSafe SwitchKit API library makes use of the Adaptive Communication Environment (ACE) library from the University of Washington, St. Louis (http://www.cs.wustl.edu/~schmidt/ACE.html) to provide the event dispatching and synchronization constructs. Any application using the threadsafe library must link with the ACE library provided as part of the distribution. In addition, the location of the ACE

shared library must be reflected in the LD_LIBRARY_PATH so that the operating system can find the ACE library when the application is invoked.

Dos and Don’ts of the ThreadSafe SwitchKit Library

Behavior of the API

As part of implementing a threadsafe API on top of the existing API, there was a layer of code that was developed to restrict access to the thread unsafe portions of the code. As a result, there are certain behaviors or peculiarities that the application developer should be aware of. Some of the issues are described below.

Sending Messages

Messages that are sent as a result of calling skts_sendMessage(), SKC_Message::tsSend() or any functions which result in messages being sent are actually queued for sending. They are sent by the thread which call skts_rcvAndDispatch()/skts_rcvAndDispatchAutoStorage() (from here on, I will only mention skts_rcvAndDispatch() by name as the functions are nearly identical). The actual send happens quickly. If the thread calling skts_rcvAndDispatch() is blocked on a socket waiting for a message from LLC, the blocking thread will wake up to send any queued up messages and then continue waiting for messages).

SwitchKit Initialization

The first SwitchKit function call made must be ***skts_enableThreadSafeAPI()***. The call to skts_enableThreadSafeAPI() must be made in the same thread that the ***skts_rcvAndDispatch()*** loop will be implemented in. No other calls to skts_rcvAndDispatch() or any switchkit receive function should be made outside the thread running the skts_rcvAndDispatch() loop. All SwitchKit handlers defined by the application are called within the context of the thread that called skts_rcvAndDispatch(). This thread should normally be blocked waiting for a message from the LLC. When a message arrives, the SwitchKit API code running in that thread will receive the message and determine which handler should receive the message. The SwitchKit API code will call the handler passing in the message just received. Applications wishing to have messages processed by other "worker" threads must arrange to get the message to the worker threads by any means at the application developer’s disposal. The exact mechanism used to get the message to the worker thread is outside the scope of SwitchKit API.

No Mixing of Threadsafe and Non-threadsafe Functions

Applications cannot mix threadsafe and standard SwitchKit functions in a single application. The thread calling skts_rcvAndDispatch() spends some portion of it’s execution path in the non-threadsafe portion of the code. If another thread attempts to access the non-threadsafe portion of the code at the same time, SwitchKit internal data will be corrupted causing your application to behave incorrectly. Therefore, multi-threaded applications must use functions beginning with "skts" or the SKC_Message::tsSend() method and no two threads may call skts_rcvAndDispatch() simultaneously.

Preferred Application Model

Multi-threaded applications developed by Dialogic customers should use the Threadsafe SwitchKit API as has been described in this chapter. The Threadsafe SwitchKit API was designed with an application model consisting of two main components:

- The main thread
- One or more worker threads.

The Main Thread

The main thread or initial thread of the process is where the application starts execution. Here, all process level initialization takes place. SwitchKit initialization should also take place within the main thread.

Table 12-1 Steps of a Main Thread Process

The main thread should minimally include:

Step	Description
1	Initialize the Threadsafe SwitchKit API by calling skts_enableThreadSafeAPI().
2	Establish connections to all LLCs. Typical applications connect to only one pair of redundant LLCs, however, in a large distributed environment, the application may be required to connection multiple LLCs. There should be one skts_createConnection() call for each LLC redundant pair in the system and each pair should be assigned a unique connection ID.

Step	Description
3	Setting the main handlers for the application. This includes the LLC connection handler (skts_setLLCConnectionHandler()), at least one default handler (skts_setDefaultHandler() or skts_pushDefaultHander()) and optionally a group handler (skts_setGroupHandler()) if performing call processing using LLC channel groups.
4	Register for SwitchKit messages. These messages are switch initiated messages which can present the current state of the switch or the current state of resources and devices on the switch. This registration is performed using skts_msgRegister(), skts_pplComponentRegister(), skts_pplTCAPRegister() or several other function.
5	Optionally, register for inbound calls being presented to LLC channel groups. This is done using the skts_watchChannelGroup()
6	Spawn worker threads. More details on worker threads will be provided in the next part of this discussion.
7	Call skts_rcvAndDispatch() or skts_rcvAndDispatchAutoStorage() in a loop. It is important that one of these two functions be called continuously and often in order to guarantee timely delivery of messages between the application and the LLC in both directions. Failure to call this function often may adversely impact the performance of the application resulting in symptoms including, but not limited to, processing lower than expected call volumes and infrequent to frequent loss of connection to LLC.

All messages sent by the LLC to the application will initially be presented to a handler or will be returned from skts_rcvAndDispatch() or skts_rcvAndDispatchAutoStorage() within the context of the thread calling the aforementioned function (in this case, the main thread). In order for the worker threads to process the message, the message itself or relevant information contained within the message, should be presented to the worker threads so that the bulk of the processing may occur within the context of the worker thread. Failure to do so will result in an application that is more complicated than a single threaded application, and less efficient since it will be unable to leverage the multi-processor platforms commonly used in application deployments.

Worker Threads

Worker threads contain the bulk of the business logic. They must receive work in the form of a message or information extracted from messages, process the work, and possibly send the next message to the CSP that is required to continue processing the transaction. These threads can be designed to perform tasks which are time-consuming and that will otherwise impact the throughput of your application. This may include performing database lookups as part of validating a caller’s right to use the services provided by your application or the level and types of services offered to a caller, ASN.1 encoding and decoding of TCAP message, or any other function which cannot be performed in a very short period of time.

Table 12-2 Initialization Sequence for Worker Threads

The next table shows the general structure of a worker thread:

Step	Description
1	Receive work from the main thread. This work can come in the form of a copy of the message received from the main thread or the relevant data from the message received.
2	Process the work.
3	Send any follow-up messages to the switch using skts_sendMessage() or SKC_Message::tsSend() or any other API that results in a message being sent.