Sabre

CCSAPI

Version 4.1.0

Aug 2001

CCSAPI Programmer's Reference Manual

Edition 4.1.0 (August 2001)

Contents

Introduction

Sabre access is very important to many products both internal and external to Sabre. These applications all need connectivity to Sabre but the connectivity is implemented in various ways. As a result there are multiple products performing the same set of functions. This is not only a duplication of development effort, but also a waste of resources. If any new features are to be added, then changes must be made to each and every product, resulting in higher maintenance costs.

The general trend that is now shifting toward 32bit applications also necessitates the development of a 32bit API to provide Sabre access. The concept of having a Common API to provide Sabre connectivity addresses all the above problems. It is also possible to introduce enough flexibility into the API to be able to add enhancements, in the future, to the API with minimum effort. This API is known as the Common Sabre API or the CSAPI. The CSAPI is available in two formats called the CCSAPI and the CSAPI OCX. This document will concentrate on the CCSAPI.

The CCSAPI is the synchronous/blocking C interface CSAPI that is available for AIX, HPUX, IRIX, Linux, Solaris, and Win32. The Win32 version of the API also includes a synchronous/blocking ActiveX/COM wrapper API called COMuxCSAPI. COMuxCSAPI provides a COM layer that, in turn, calls the C interface CCSAPI. The COMuxCSAPI was primarily designed for use in MS Active Server Page technology but may be used by any ActiveX/COM compatible technology including MS Visual Basic and MS Visual C++. The CCSAPI is designed for high throughput applications, including severs and web sites, by virtue of the fact that it has been written in very efficient and portable C++ code.

Document Purpose

The purpose of this document is to provide a description of the C interface to CSAPI (CCSAPI) and provide information as to how to use the API on each of the platforms that it is available for. For further information regarding Sabre communications please consult the document entitled “Sabre Specification - Sabre Programmer Reference Manual” or, in shorthand, the “Sabre Spec”.

This product provides a C based interface to Sabre, CCSAPI, allowing connectivity to Sabre via a wide range of application development environments using a variety of operating systems including AIX, HPUX (10.20 RISC 2.0), IRIX, Linux (RedHat 6.2), Solaris (2.6), and Win32 (Win95, Win98, NT4). The ActiveX/COM based interface to Sabre, COMuxCSAPI - that is provided with the Win32 distribution, allows connection to Sabre via the wide range of application development environments that have the facility to use ActiveX COM Controls.

Product Overview

The CCSAPI (C Interface Common Sabre Application Programming Interface) provides connectivity to Sabre via a synchronous/blocking C interface API and also, in the case of the Win32 distribution, a synchronous/blocking COM (Component Object Model) interface. The CCSAPI includes the following features:

- Common Interface
- Heterogeneous Network Environments
- Synchronous Communication
- Flexibility
- Reusability
- Portability
- Machine Independent Solution

Supported Platforms and Dependencies

The CCSAPI is supported on several platforms including:

- AIX
- HPUX (10.20 RISC 2.0)
- IRIX
- Linux (RedHat 6.2)
- Solaris (2.6)
- Win32 (Win95, Win98, NT4, Win2000)

Architecture

The CCSAPI is based on a two-layer architecture, each layer handling a specific functionality. The two layers are:

- Connection Layer – This layer consists of the interface into CCSAPI that is visible to the Application Layer that is the customer's application. The Connection Layer handles all of the connection specific details by providing a common interface to the Application Layer and dynamically using the Services Layer to connect to Sabre as and when required. Currently the primary interface to CCSAPI is the C interface that is exported for several different Operating Systems. There is also an ActiveX/COM interface provided with the Win32 distribution of CCSAPI.

- Services Layer – This Layer consists of modules that are dynamically loaded at run time by the Connection Layer depending upon the configuration of the Connection Layer and which Sabre Services are to be used. These modules are highly specific to the host with which they wish to communicate. For example, currently there are three possible Services that provide Sabre connectivity; OSG, OFEP, and HSSP. Each of these Services requires a unique implementation of their Services interface through the Services Layer. The Services Layer then provides the Connection Layer with a common interface for accessing Sabre through any of these Services.

The Application Layer

The Application Layer is written by the customer and is not provided as part of the CCSAPI distribution. This Layer is not constrained to being a single physical tier, it could be part of an n-

tier application. The Application Layer handles the session management functionality for the system as a whole. Session management is a level of abstraction above the Connection Layer that actually provides the Communications API. The Application Layer is where business logic functionality and business transaction encapsulation logic is based. This layer communicates to Sabre through the Connection Layer. Note that the ActiveX/COM version of the Connection Layer is only available with the Win32 distribution of CCSAPI. The C Interface version of the Connection Layer is available with all distributions of CCSAPI.

The Connection Layer

The Connection Layer is the CCSAPI product. This Layer handles all of the communication specific details through a common interface to the Application Layer. A client would use this layer as the API with which their application would communicate with Sabre. This layer is responsible for instantiating the final host specific Services Layer. The host session parameters are recorded in a configuration file. The appropriate host Service Layer Service Provider is then instantiated as per the settings in the configuration file. The C Interface API is specified in csapi_structs.h file and the Win32 C++ ActiveX/COM Interface API is specified in the COMuxCSAPI.h file.

The Services Layer

The Services Layer consists of host specific modules also called Service Providers which will be loaded as and when they are required. These are highly specific to the host with which they wish to communicate. This host can be either an end system like Sabre or an intermediate server that communicates with and end system such as an Open Systems Gateway (OSG). An advantage of this layer is that a client can communicate either to a gateway or an end system like Sabre using the same interface. This provides a single client application the ability to talk to the gateway of their choice without any change to their Application Layer logic. The interface to this layer is not published or provided to customers in order to maintain uniformity of access at the Connection Layer.

Service Layer Service Providers

The Service Providers currently supported are:

1) Open Front End Processor (OFEP) service provider:
   Implements the Sabre IP front end protocol. Please contact the OFEP provisioning group for LineIATA allocation.

2) Open Systems Gateway (OSG) service provider:
   Implements the OSG X.25 protocol. The OSG implements the Sabre specifications over an X.25 network for host communication and provides a socket level interface on the LAN. The user is responsible for the OSG gateway infrastructure.

3) Host Session Service Protocol (HSSP) service provider:
   Implements the Sabre HSSP IP protocol directly to the native Sabre TPF mainframe system via the NOFEP system bypassing the OFEP and the Microvax Front End (MFE) system. At this time this system is not in production and is being tested. In the future, applications using the CCSAPI will be able to seamlessly migrate to this service provider without any design or code changes.

CCSAPI Prerequisites

Ensure that the customer site has connectivity to Sabre either through X.25, ALC, or TCP/IP. On a Win32 site this may be confirmed by ensuring that Sabre for Windows is able to connect to Sabre, on a UNIX site the Sabre connectivity provider should be able to confirm connectivity is correctly installed.

CCSAPI C Interface

Note that the C interface to CCSAPI is the standard interface and, unlike the case of the ActiveX/COM interface, it is available on all platforms and not just Win32.

Exported Functions

The functions exported in the C interface to CCSAPI.dll, in the case of Win32, and LibCSAPI.so, in the case of UNIX, are specified in the csapi_structs.h file. The exported functions are listed below:

ConnectToHost

Prototype: int **ConnectToHost**(void** ppvHandle, char* pcRoute)

Description: Call this function to establish a communication session with the Host.

Parameters: The ppvHandle parameter is returned after the function is executed and specifies the new instance address of the communication object inside CSAPI. It is used as an identifier for the current instance and is passed to all subsequent function calls as a parameter
The pcRoute parameter specifies a Route defined in the csapi.cfg file.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

ConnectToSpecifiedHost

Prototype: int **ConnectToSpecifiedHost**(void **ppvHandle, char *pcRoute, SESSION_DATA *SessionData, LOG_SETTINGS *Settings, SP_OPTION_ARRAY SPOptionArray, int NbrSPOptions)

Description: Call this function to establish a communication session with a specified Host. Note that this function is currently only supported by the HSSP Service Provider. Please see the section below titled 'Method for establishing an HSSP session' for further information on how this function should be used with HSSP.

Parameters: The ppvHandle parameter is returned after the function is executed and specifies the new instance address of the communication object inside CSAPI. It is used as an identifier for the current instance and is passed to all subsequent function calls as a parameter
The pcRoute parameter specifies a Route defined in the csapi.cfg file.
The SESSION_DATA structure may be used at runtime to override the settings defined by the route defined in the csapi.cfg file. This would primarily be used to achieve session sharing by using the returned

values from GetSessionData to connect to a pre-existing session. If you do not intend to use this structure, it is best to pass in a NULL pointer instead of an empty structure. If you have to pass in an empty structure, make sure it is zero (NULL) initialized or unpredictable behavior may result.
The LOG_SETTINGS parameter can be used to override the log settings described by your route for this session only. This is similar to calling the SetSessionLogging function. If you do not intend to use this structure, it is best to pass in a NULL pointer instead of an empty structure. If you have to pass in an empty structure, make sure it is zero (NULL) initialized or unpredictable behavior may result.
The SP_OPTION_ARRAY parameter combined with the NbrSPOptions parameter can be used to pass in one or more options to the service provider at the time of connection. The NbrSPOptions parameter corresponds to the number of option in the array. Please see the section below titled 'Method for establishing an HSSP session' for further information on how these parameters should be used with HSSP.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

GetSessionData

Prototype: int **GetSessionData**(void* pvHandle, , SESSION_DATA* pSessionData)

Description: Once a session has been successfully established, the GetSessionData function will serialize the session into the SESSION_DATA structure. The data in this structure can be fed to the ConnectToSpecifiedHost function to establish another connection to a pre-existing host session. Since the options data returned in the structure may potentially change after each ReceiveData or ReceiveDataWithTimeout it is advisable to call GetSessionData after each Receive to maintain current session data.

Parameters: The pvHandle parameter specifies the current instance address of the communication object inside CSAPI.
The pSessionData SESSION_DATA pointer parameter returns the SESSION_DATA structure that has been generated when the call to Connect was completed.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

ConnectToHostGetSid

Prototype: int **ConnectToHostGetSid**(void** ppvHandle, char* pcRoute, char* pcSid)

Description: Call this function to establish a communication session with the Host and get the current Session ID, or Line IA TA, in one single API function call.

Parameters: The parameters are as described above in the ConnectToHost and GetSid functions.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

GetSid

Prototype: int **GetSid**(void* pvHandle, char* pcSid)

Description: Call this function to get the current Session ID or LineIATA.

Parameters: The pvHandle parameter specifies the current instance address of the communication object inside CSAPI.
The pcSid parameter is returned after the function is executed and specifies the 6 character LineIATA currently being used by the communication session instance.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

SendData

Prototype: int **SendData**(void* pvHandle, char cState, char* pcData, int* piDataLen)

Description: Call this function to send data to the Host.

Parameters: The pvHandle parameter specifies the current instance address of the communication object inside CSAPI.
The cState parameter is a currently unused future enhancement.
The pcData parameter is the address of an array of characters that are to be sent to the Host.
The piDataLen parameter is a pointer to an int that holds the number of characters to be sent to the Host.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

SendWipe

Prototype: int **SendWipe**(void* pvHandle, char cState, char* pcData, int* piDataLen)

Description: Call this function to wipe the socket buffer before sending data to the Host in one single API function call. Note: Do not abuse this function as it will clear all messages on the socket including OFEP control messages. This function is to be used with care and is only provided as an aid to clear the socket if data from previous transactions are resident due to timeouts or application errors.

Parameters: The parameters are as described above for the SendData function.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

ReceiveData

Prototype: int **ReceiveData**(void* pvHandle, char** ppcData, int* piDataLen)

Description: Call this function to get the data in the socket buffer from the Host.

Parameters: The pvHandle parameter contains the current session handle.
The pcData parameter is returned when the function executes and contains the data from the Host.
The piDataLen parameter is returned when the function executes and contains the number of characters returned from the Host.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

ReceiveDataWithTimeout

Prototype: int **ReceiveDataWithTimeout**(void* pvHandle, char** ppcData, int* piDataLen, int iTimeout)

Description: Call this function to get the data in the socket buffer from the Host, waiting for the iTimeout time for data to arrive if there is currently no data. Note: the value passed in this parameter overrides the READTIMEOUT csapi.cfg file service definition parameter.

Parameters: The parameters are as described above for the ReceiveData function except for the iTimeout parameter which indicates a timeout in seconds with a minimum of 1 and a maximum of 200.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

SetSessionLogging

Prototype: int **SetSessionLogging**(void* pvHandle, LOG_SETTINGS* pLogSettings)

Description: Default log settings are defined in the csapi.cfg file with the entries DEBUG, ERRORLOGGING, LOGLEVEL, and LOGFILE. These settings will be the default settings for every session created from the defined route. SetSessionLogging can be used to modify the default settings at run time for the given session. A typical scenario would be defaulting logging to off, or to errors only. If problems are being encountered for a specific session, logging can be turned on for that session for diagnostic purposes and then reset to the default when you're done. To reset log settings to the defualt call SetSessionLogging with a NULL pointer in place of the LOG_SETTINGS structure.

Parameters: The pvHandle parameter contains the current session handle.
The pLogSettings parameter is a pointer to a modified LOG_SETTINGS structure that in turn overrides the log settings for the session as configured in the csapi.cfg file.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

GetLogSettings

Prototype: int **GetLogSettings**(void* pvHandle, LOG_SETTINGS* pLogSettings)

Description: The GetLogSettings function would be called to query the session specified by pvHandle for the currently configured error logging settings, these would be returned in the LOG_SETTINGS structure pLogSettings.

Parameters: The pvHandle parameter is a pointer to the current session handle. The pLogSettings parameter is a pointer to the structure that will return the current log settings of the current session.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

SetServiceProviderOption

Prototype: int **SetServiceProviderOption**(void* pvHandle, int iSPOptiontype, char* pcOptionValue)

Description: This function was originally implemented in order to allow HSSP options to be set, however there are not currently any service provider options that are valid to be set after the connection to Sabre has been made. The function was made to be generic so that other service providers could use the function to set service provider specific options as required. There are currently no HSSP options that need to be set after the call to ConnectToHost. Since this is a generic function it has been left in the API for future use. Note: This function is currently only implemented for use with the HSSP Service provider.

Parameters: The pvHandle parameter contains the current session handle.
The iSPOptiontype parameter refers to the SP_OPTIONS enumerated type values in the csapi_structs.h file.
The pcOptionValue parameter refers to the actual service provider option string.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

GetServiceProviderOption

Prototype: int **GetServiceProviderOption**(void* pvHandle, int iSPOptiontype, char* pcOptionValue)

Description: This function was originally implemented in order to allow HSSP options to be retrieved. The function was made to be generic so that other service providers could use the function to get their service provider options. Since this is a generic function it has been left in the API for future use. Note: This function is currently only implemented for use with the HSSP Service provider.

Parameters: The pvHandle parameter contains the current session handle.
The iSPOptiontype parameter refers to the SP_OPTIONS enumerated

type values in the csapi_structs.h file.
The pcOptionValue parameter refers to the actual service provider option string that will be returned.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

GetTotalMessageLength

Prototype: int **GetTotalMessageLength**(void* pvHandle, STAT_TYPE StatisticType, int* piNbrBytes)

Description: This function is for statistics gathering purposes. The length of the payload without headers is still passed into the Send functions or returned from the Receive functions as is normal if this function is called.

Parameters: The pvHandle parameter contains the current session handle.
The StatisticType parameter specifies a particular item from the STAT_TYPE enumerated type values in the csapi_structs.h file.
The piNbrBytes parameter is returned containing a value that represents the actual number of bytes sent or received including header information.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

ActiveConnections

Prototype: int **ActiveConnections**(int* piNbrActiveConnections)

Description: Version 4.x of the CCSAPI uses a shared control block to keep track of active sessions within the process. This means that if ConnectToSpecifiedHost is called 4 times within the same process to connect to the same host session then the first call will establish the connection to the host session and the following calls will simply be given the handle to the original connection. ActiveConnections can be used to monitor the number of connections that are active within the process. In the above scenario where ConnectToSpecifiedHost was called 4 times to connect to the same host session ActiveConnections would return 1, since only one physical connection is active in this process. If each connection was made to a different host session then ActiveConnections would return 4.

Parameters: The piNbrActiveConnections parameter is returned containing a value that represents number of active connections to Sabre from the current process.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

DisconnectFromHost

Prototype: int **DisconnectFromHost**(void* pvHandle)

Description: Call this function to disconnect the current session from the Host. The

pvHandle parameter contains the current session handle. To get detailed information regarding the returned int see the return codes section of this document.

Parameters: The pvHandle parameter contains the current session handle.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

Disconnect

Prototype: int **Disconnect** (void* pvHandle)

Description: Note: This function is currently only implemented for use with the HSSP Service provider. The Disconnect function is similar to the DisconnectFromHost function. DisconnectFromHost first sends a termination packet to the host and then closes the tcpip socket. Disconnect differs in that no termination packet is sent. The socket is simply closed. In a session sharing scenario this would be used to close a connection to a session without terminating the host session that another connection may be using.

Parameters: The pvHandle parameter contains the current session handle.

Returns: To get detailed information regarding the returned int see the return codes section of this document.

Return Value and Error Codes

General Error Codes

-1 ERR Generic Error: An unspecified error occurred.
0 NOERR No Error: Success, there was no error.
2 SOCKETERR Socket Error: An operation is being invoked on a non connected socket. Improper socket type is specified in the configuration file. Occurs if the application uses the API after disconnecting the connection or on a unconnected session.
3 CONNECTERR Connect Error: An error occurred during the connect. The details are recorded in the log file. It could be due to invalid load balancer entries, network related problems or duplicate TA usage.
4 BADIPADDR Bad IP Address: Bad IP Address specified in csapi.cfg file.
5 BADPORT Bad Port: Bad IP Port specified in csapi.cfg file.
6 BADHOSTNAME Bad Host Name: Bad Host Name specified in the csapi.cfg file: A DNS name resolution “gethostbyname” function call returned a null value which means the hostname is not valid on the IP network. Check the parameter in the config file.
7 RECEIVEERR Receive Error: This happens when the TCP/IP receive operation failed. A global error no is logged in the log file. See note on global error number below.
8 SENDERR Send Error: This happens when the TCP/IP send operation failed. A global error no is logged in the log file. See note on global error number below.
12 CFGFILEERR Configuration File Error: Error reading the configuration file csapi.cfg: The configuration file, csapi.cfg, is either not in the current directory, not in the Windows directory on Win32 systems determined by the GetWindowsDirectory() Win32 function call, or not in the location specified by the CSAPI_CFG_FILE environment variable on UNIX systems.
13 INVALIDSERVICETYPE Invalid Service Type Error: Bad service type specified in the configuration file csapi.cfg: The service provider name associated with the route name parameter in the ConnectToHost call is not found in the configuration file.
20 NEWERR New Memory Allocation Error: New operation failed or memory allocation error (e.g. out of memory).
21 DLINKERR Dynamic linking error. The shared objects could not be loaded at runtime. Usually due to invalid LD_LIBRARY_PATH or invalid shared objets/dynamic link libraries (dlls).
22 MUTEXERR Mutex creation error. This will be seen only if the system is out of memory and no mutex variables can be created by the thread subsystem.
23 TEXTDATA Text message (for SABRE Binary Transport Protocol only).
25 KEEPALIVETIMEOUT Keep Alive Packet Timeout: Failed to receive a Keep Alive packet within the Keep Alive timeout period. The connection is assumed to be down.
26 OVERRIDECFGERR Configuration File Override Error: There was an error while attempting to override the default configuration for the session with the data contained in the SESSION_DATA structure.
27 SESSIONCBERR Session Control Block Error: An error has occurred in the Session Control Block.

HSSP Specific Error Codes

9 STREAMHDRERR HSSP Stream Protocol Error: The HSSP Stream Header has returned an error code from the host. There are currently no Stream Header Errors defined by the host for HSSP. Reserved for future use.
10 SESSHDRERR HSSP Session Protocol Error: The HSSP Session Header has returned an error code from the host.

OFEP Specific Error Codes

11 DATAERR Data Error: Returned if there is a packet format error in a multi-packet message.

14 LBERROR OFEP Load Balancer Error: Occurs if the name lookup for the Load Balancer fails. Check the configuration file. Check the log file for more details. The load balancer may not respond in time, try bumping the READTIMEOUT and the CONNECTTIMEOUT up on high latency networks or during periods of high utilization. This error may occur if the Load Balancer sends an error packet back and is logged.

15 UNKNOWNLBREPLY Unknown Reply Error: An unrecognized response was received from the OFEP Load Balancer.

16 TIMEOUT TCP/IP Timeout Error: TCP read or write operation timed out. Try the operation again after a brief delay.

18 OFEPERR OFEP Error: This error occurs in any of the following cases:

1)The OFEP has sent a packet type 4 and has indicated a fatal error. Read the Sabre IP specification document available at http://sabreweb.sabre.com/sabreip for more details.

2)The OFEP has sent a type 5 shutdown packet.

3)The OFEP sent a packet with an invalid packet type. This occurs usually due to invalid data read from the socket. For instance the length to read is invalid and the OFEP header cannot be interpreted properly.

24 LBSTANDBY Load Balancer Standby Error: The primary OFEP Load Balancer is in standby mode.

OSG Specific Error Codes

Currently there are no OSG specific error codes

Global Error Number

When a system call or TCP/IP call fails on a UNIX based operating system the global error number is recorded in the log file. A UNIX “grep” operation on the error number in “/usr/include/sys/errno.h” should identify the cause of the problem.

Configuration File

The CCSAPI configuration file is called csapi.cfg. The location of this file is dependant upon the operating system that is currently being used and is explained in the chapters addressing the configuration file location on Win32 and UNIX respectively.

Configuration File Syntax

A character sequence of “##” may be used to record comments.

Stand alone parameter entries have the following syntax:

```
NAME = "value"
```

A parameter section entry has the following syntax:

```
{section name { NAME = "value" NAME = "value"     } }
```

The end of file is indicated by a sequence of %% characters.

The following three entries are used only by the asynchronous mode libraries:

```
Turn Debugging on or off for the CSAConn class
##TRACE =  YES or NO
TRACE = "YES"
## Name of log file
LOGFILE = "csapi.log"
## Log Level 1200 for all levels , 35 for only errors
LOGLEVEL = "1200"
```

Route Names

A route name is the parameter passed into the connect call. It consists of at lease one service parameter and an optional session ID parameter. The service value specifies the name of service provider to use. The optional session ID parameter given by "sid" is used if a fixed TA or session is required. If sid is omitted host pooled TA's will be used. Below are 5 examples of different route name configurations, their names imply the type of configuration, but the name may be chosen by the customer, i.e. POOLEDOFEPSABRE could be renamed ROUTEONE or whatever is required:

```
{ POOLEDOFEPSABRE  {  service = "pooledofep" }  }
{ POOLEDOSGSABRE { service = "osg" } }
{ FIXEDOFEPSABRE  { service = "fixedofep" sid="94EF02"  }  }
{ FIXEDOSGSABRE  { service = "osg" sid="F20606" }  }
{ FIXEDHSSPSABRE  { service = "hssp" sid="C20606" }  }
```

The API will search the configuration file for the route name. It will identify the service provider to use as given by the service parameter. It will then search the configuration file for the service provider section to get the values of its parameters. The parameters for the service providers are described below.

General Service Provider Options:

These options are set under the service provider sections:

READTIMEOUT:	Sets the time out value for a read operation (The max value is 200 secs and minimum value is 0).
HOSTTYPE:	"OFEP","HSSP" , or "OSG" : specifies the host type to use for the loader class.
SOCKETTYPE:	Specifies either "TCP" or "UDP".
PORTNUMBER:	Port number to connect to.
EBCDIC2ASCII:	Specifies if translation from SABRE EBCIDIC to ASCII translation is required. Value can be "YES" or "NO ".
CONNECTTIMEOUT:	Specified the time out value for a connect call.
DEBUG:	Turns service provider logging to on or off. Values may be "YES" or "NO".
ERRORLOGGING:	Specifies the logging mode, values may be:
"LOGTOFILE"	prints messages to a file specified by LOGFILE parameter.
"LOGTOSCREEN"	prints message to stdout.

"LOGTOBOTH"	prints messages to both.
LOGFILE:	Name of log file
LOGLEVEL:	Level of logging. The default level is "ERROR". The values may be one of the following:
"INFO"	detailed logging
"WARNING"	logs both error and warning messages
"ERROR"	logs only error messages .This is the default
SKIPLEADINGCHARS:	Specifies number of leading characters to skip in the host data buffer before delivery to application. Values may be: 3 which skips the 3 byte LineIATA field 5 which skips both LineIATA and following c1 c2 characters
TABLETYPE:	Starting from version 3.0 we now have the ability to load custom character conversion tables. The default value is "NORMAL". Currently only travelocities conversion table is supported when the value for this parameter is set to "TRAVELOCITY".

OFEP Specific Service Provider Options

The following options are used in OFEP service provider definitions only:

VERSION:	Version number of the protocol, currently 1.
SERVICETYPE:	Can be either LNIATA or CLASS. If LNIATA then a sid has to be specified. If CLASS then the class name must be specified in the SERVICETYPE QUALIFIER parameter.
SERVICETYPE QUALIFIER:	Specifies the class name configured in the OFEP. This is used when the SERVICETYPE is set to CLASS.
PRILOADBALANCER:	Name address of the primary Load Balancer.
SECLOADBALANCER:	name address of the secondary Load Balancer.
KEEPALIVE:	If "YES" the OFEP host will check inactive sessions with a keep alive packet and is preferable. Abnormally terminated clients will be reclaimed by the OFEP as this keep alive packet operation on such a client will fail. Default is "NO.
NORECEIVETHREAD:	"YES" should be used for the Synchronous version.
TABLETYPE:	"NORMAL" should be used unless you have a custom table built specifically for you in the API. This is the default.
LOGLEVEL:	INFO,WARNING ,ERROR INFO will cause everything to be logged. WARNING will log both ERRORS and WARNINGS.

OSG Specific Service Provider Options

The following options are used in OFEP service provider definitions only:

LOGLEVEL:	Specifies the logging level. The value can be "10000" which prints all trace messages. To see only critical errors set value to "35".
HOSTADD:	Specifies the IP address of the OSG gateway.
LOCALHOST:	Specifies the local host name in case the HOST variable is not set.

HSSP Specific Service Provider Options

The following options are used in HSSP service provider definitions only:

HOSTNAME:	Specifies primary HSSP host IP address.
SESSION_VERSION:	Specifies session version currently "1".
STREAM_VERSION:	Specifies stream version currently "1".

TA_TYPE:	Specifies LineIATA allocation method: FIXED, HOSTPOOLED, AGENCYPOOLED.
DEVICE_TYPE	Specifies terminal or printer: TERMINAL, PRINTER.
PROFILENAME	Specifies profile name for TAM sessions or pseudo city code for AGENCYPOOLED sessions.
PROFILEKEY	Specifies profile key for access to TAM profiles requiring a key.

Example csapi.cfg File

Below is an example csapi.cfg file:

```
##Route names
{ SABRE  {  service = "pooledofep" }  }
{ OSGSABRE { service = "osg" } }
{ FIXEDOFEPSABRE  { service = "fixedofep" sid="000002"  }  }
{ FIXEDHSSPSABRE  { service = "hssp" sid="000004" }  }
##Service definitions
{ fixedofep {
      READTIMEOUT = "2"
      HOSTTYPE="OFEP"
      SOCKETTYPE="TCP"
      PRILOADBALANCER="lb1.dcs.amrcorp.com"
      SECLOADBALANCER="lb2.dcs.amrcorp.com"
      PORTNUMBER="12001"
      VERSION="1"
      SERVICETYPE="LNIATA"
      EBCDIC2ASCII="YES"
      READTIMEOUT="3"
      CONNECTTIMEOUT="3"
      KEEPALIVE="YES"
      DEBUG="YES"
      ERRORLOGGING="LOGTOFILE"
      LOGFILE="OfepLog.txt"
      SKIPLEADINGCHARS="5" }}
{ pooledofep {
      READTIMEOUT = "4"
      HOSTTYPE="OFEP"
      SOCKETTYPE="TCP"
      PRILOADBALANCER="lb1.dcs.amrcorp.com"
      SECLOADBALANCER="lb2.dcs.amrcorp.com"
      PORTNUMBER="12001"
      VERSION="1"
      SERVICETYPE="CLASS"
      SERVICETYPEQUALIFIER="ENTERCLASSNAMEHERE"
      EBCDIC2ASCII="YES"
      READTIMEOUT="3"
      CONNECTTIMEOUT="3"
      KEEPALIVE="YES"
      DEBUG="NO"
      ERRORLOGGING="LOGTOFILE"
      LOGFILE="OfepLog.txt"
      SKIPLEADINGCHARS="5" }}

{hssp {
      HOSTTYPE="HSSP"
      LOGFILE="HsspLog.txt"
      SOCKETTYPE="TCP"
      HOSTNAME="192.161.103.105"
      HOST2="192.161.104.106"
      PORTNUMBER="6030"
```

```
      READTIMEOUT="15"
      CONNECTTIMEOUT="15"
      SESSION_VERSION="1"
      STREAM_VERSION="1"
      DEBUG="YES"
      EBCDIC2ASCII="YES"
      ERRORLOGGING="LOGTOFILE"
        }}
{osg {
      LOGFILE="osg.log"
      HOSTTYPE="OSG"
      SOCKETTYPE="TCP"
      PORTNUMBER="1413"
      EBCDIC2ASCII="YES"
      READTIMEOUT="5"
      CONNECTTIMEOUT="5"
      DEBUG="YES"
      LOGLEVEL="10000"
      HOSTADD="192.161.206.130"
      LOCALHOST="rodeo"
      SKIPLEADINGCHARS="5"}}
%%
```

Character Translation

The CCSAPI provides the facility for the user to modify the ASCII to Sabre and Sabre to ASCII character set translation mappings for data being sent to Sabre and being received from Sabre respectively.

Translation Overview

Sabre Host Character Sets

The Sabre system originally used a 6-bit character set that allowed a maximum of 64 characters to be defined. Simply enough, this character set is known as the Sabre Character Set. Later, a need for more than 64 characters became necessary, but the 6-bit restriction was still in place. This problem was resolved by creating a second 6-bit character set called the Extended Sabre Character Set combined with the use of an escape character to differentiate between the Sabre Character Set and its Extended cousin. To clarify, both character sets contain 64 characters, represented by the same hexadecimal values 0x00 through 0x3F. A Sabre character can be referenced in the normal manner by its hex value. An Extended Sabre Character is referenced by its hex value also, however, it is prepended by an escape character. This means that it takes two characters to represent a single Extended Sabre Character. More information on these character sets can be found in the document entitled “Sabre Specification - Programmer’s Reference Manual”.

Sabre Host Front End Processors

Client applications interface with the Sabre system through Front End Processors (FEP). These FEPs communicate with the client application through the use of the 8-bit EBCDIC character set. The FEPs translate EBCDIC characters into Sabre characters and Extended Sabre characters. Again, it takes a single EBCDIC character to represent a single Sabre character, and it takes two EBCDIC characters to represent a single Extended Sabre character. More information on this can be found in the document entitled “Sabre Specification - Programmer’s Reference Manual”.

Client Character Sets

Today, most clients use the ASCII character set, though for the purposes of CCSAPI this is not strictly required. CCSAPI uses hexadecimal values to represent character values, so any 8-bit character set can be used. Default translation is between ASCII and EBCDIC however.

Translation Configuration

CCSAPI can translate any 8-bit character set used by the client both into and out of the proprietary EBCDIC data stream used by the Front End Processors. In most applications this feature greatly reduces the burden on the client applications programmer to deal with Sabre's proprietary character sets. In a few cases the client application will want to handle the data stream more directly. In those rare cases it is possible to disable CCSAPI's translation feature on a Service by Service basis. Please refer to the chapter on CCSAPI Configuration for a definition of Services.

Enabling and disabling character translation in CCSAPI

CCSAPI allows the character translation feature to be enabled or disabled. This allows the calling application the flexibility to interpret the raw data streams itself, or allow CCSAPI to take care of this complex task instead.

The EBCDIC2ASCII entry in the Services section of the CCSAPI configuration file controls this. EBCDIC2ASCII="YES" enables the translation feature and EBCDIC2ASCII="NO" disables it.

Using the default character translation tables

In order for the translation process to work correctly CCSAPI needs to know exactly what client character should be mapped to what host character(s) and vice versa. Sets of these character mappings describe a 'translation table'. Two tables are necessary, one for Client to Host mappings, and one for Host to Client mappings. These two tables together describe a translation table set. The TABLETYPE entry in the Services section of the CCSAPI configuration file controls which translation table set is to be used.

CCSAPI has a default translation table set defined for typical use. If TABLETYPE="NORMAL" is specified, or the TABLETYPE entry is completely omitted, then the default translation table set will be used. Please refer to the normal.map file included in the CCSAPI distribution package for details on how each character is mapped in the NORMAL translation table set.

CCSAPI also has a secondary translation table set defined for Travelocity's use. A number of CCSAPI's customers use these Travelocity character mappings so these tables have been included as a pre-defined table set within CCSAPI. To use this translation table set use the TABLETYPE="TRAVELOCITY" setting. Please refer to the Travelocity.map file included in the CCSAPI distribution package for details on how each character is mapped in the TRAVELOCITY translation table set.

Overriding the default character translation process

For some applications the default translation table sets will be insufficient. Therefore a method for defining custom translation table sets is available. Use the TABLETYPE entry in the Services section of the CCSAPI configuration file for this purpose. The format of this entry is:

TABLETYPE="[NORMAL] | [TRAVELOCITY] | [mapfilename]"

The NORMAL and TRAVELOCITY options were described in the section above. The mapfilename option will be described here.

The mapfilename option tells CCSAPI to look for a map file defining a custom character table set. The file name may be fully or partially qualified. If no path is provided, the current working directory will be used.

This map file can, but is not required to, define all character mappings that you will use. This is an important distinction. If the NORMAL table set works well for you, with the exception of only a few of the character mappings, then this file only needs to describe the exceptions. To be specific, when a map file is specified then CCSAPI always loads the default NORMAL translation table set first, and then reads the map file for overrides to those settings. If there is a need to override all of the default character mappings then that can of course be done by describing every mapping in the map file.

The file is divided into two sections defining each of the two translation tables required. One section to describe Client to Host mappings and the other to describe Host to Client mappings.

The file itself should be an ASCII text file following the format described below.

Sabre Escape Character

The character that the Front End Processors use as an escape character in an EBCDIC stream is 0x5D. As a side note, Sabre programmers often refer to this escape character as the DLE character. When dealing with a Sabre Help Desk or Sabre Programmer it will most usually be referred to as a DLE.

Map File Comments

C++ style comments may be used in the map file wherever desired. Use '//' (Two forward slashes) to indicate a comment. The CCSAPI map file parser will ignore everything on a line following the comment indicator, the comment indicator included.

Example:

```
0x41=0xC1 //mapping ASCII 'A' to Sabre 'A'
```

Section Headers and Entries

The file is separated into two sections defined by the following section headers:

[CLIENT TO HOST CHARACTER MAP]

This section defines Client to Host character mappings. Since the Client character set is described as an 8-bit character set there can be a maximum of 256 mappings described in this section. If a client character is mapped more than once then the last mapping defined will be used. Entries in this section will follow the pattern of:

```
0x##=0x##[,0x##]
```

Where # represents a hexadecimal digit. The 0x##=0x## format will represent mapping from a single client character to a single EBCDIC character. The 0x##=0x##,0x## format will represent a mapping from a single EBCDIC character to a two character EBCDIC stream. Under usual circumstances, the first EBCDIC character will be an escape character telling the Sabre system to translate the second EBCDIC character into an Extended Sabre character.

Examples:

```
0x41=0xC1 //mapping ASCII 'A' to Sabre 'A'
```

0x41=0xC1 maps the ASCII character 0x41, an 'A', to the EBCDIC character 0xC1. The Sabre Front End Processor then translates the

EBCDIC 0xC1 into the Standard Sabre 0x31 character, which is an ‘A’.

0x61=0x5D, 0xC1 //mapping ASCII ‘a’ to Sabre ‘a’

0x61=0x5D,0xC1 maps the ASCII character 0x61, an ‘a’, to the EBCDIC two byte character set 0x5D and 0xC1. The Sabre Front End Processor recognizes the 0x5D as an escape character, telling it to interpret the following character, 0xC1, from the Extended Sabre character set instead of from the Standard Sabre character set. In this case, 0xC1 also translates to 0x31, but on the Extended Sabre character set 0xC1 translates to an ‘a’ rather than an ‘A’.

[HOST TO CLIENT CHARACTER MAP].

This section defines Host to Client character mappings. The Front End Processors (FEP) use the EBCDIC character set, which is an 8-bit character set. This would imply that this section could contain 256 entries. However, keep in mind that the FEP also uses an escape character to signal that a particular host character originated from the Extended Sabre Character Set. In practice you will seldom define more than 128 characters as described in the two Sabre character sets, but it is theoretically possible to define 256 non-escaped character mappings and 256 escaped character mappings for a theoretical maximum of 512 character mappings in this section. Realistically, if 512 entries were to be defined then it would necessarily to duplicate mappings to the 256 client characters, so when the escaped and non-escaped host character mappings are combined their numbers should not exceed the 256 possible client characters available. Entries in this section will follow the pattern of:

0x##[,0x##]=0x##

Where the # represents a hexadecimal digit. The 0x##=0x## format will represent mapping a single non-escaped host character to a single client character. The 0x##,0x##=0x## format will represent mapping an escaped host character sequence to a single client character.

Examples:

0xC1=0x41 //mapping Sabre ‘A’ to ASCII ‘A’.

The Sabre Front End Processor translates the Sabre 0x31 character, an ‘A’, into an EBCDIC 0xC1. 0xC1=0x41 maps the EBCDIC 0xC1 to an ASCII 0x41, an ‘A’ in ASCII.

0x5D,0xC1=0x61 //mapping Sabre ‘a’ to ASCII ‘a’.

The Sabre Front End Processor translates the Extended Sabre 0x31 character, an ‘a’, into an EBCDIC 0x5D, 0xC1 stream. 0x5D,0xC1=0x41 maps the EBCDIC 0x5D, 0xC1 stream to an ASCII 0x61, an ‘a’ in ASCII.

Special Sabre Characters

GDS Characters

The following ASCII values are defined in csapi_defines.h and should be used to download GDS/SDS streams. Please visit http://sds.sabre.com for more information about GDS/SDS.

Constant	ASCII hex values	EBCDIC
GDS_ITEM_DELIM_CHAR	0x86	0x50
GDS_SEG_DELIM_CHAR	0x89	0x5D50
GDS_END_OF_REPEAT_DATA	0x88	0x5D61

GDS end of message is represented as two GDS_SEG_DELIM_CHAR in sequence
ie 0x89 0x89

Sabre Characters

Sabre CHANGE character = '@' or 0xA4
Sabre ENDITEM character = '\'
Sabre CROSSLORRAINE character = “’” or 0x87

C Interface Programming Examples

C Example 1

The following is a simple C language console based application illustrating the use of the C Interface to CSAPI to allow the user to communicate with Sabre:

```
#include <time.h>
#include <sys/timeb.h>

#include <errno.h>
#include <sys/types.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>


#include "csapi_structs.h"
#include "csapi_defines.h"

int main( int argc,char**argv)
{
   int               GDS = 0;
   char              fn[] = "SABRE";

   static char       fname[] = "CsapiSyncTest()";
   char              request[1024];

   void*             handle;
   char*             InData;
   char              prtData[32767];
   int               InDataLen, OutDataLen;
   int               result, quit, firsttime;
   char              State = ' ';
   char              sid[7];
   char              lta[7];
   int               i, loop;

   //time specific variables
   struct _timeb tstruct1, tstruct2;
   int time_total;

   loop = 100; //number of repetitions through send "1DFWLAX"
   time_total = 0;

   memset( request, 0x00, sizeof(request) );
   memset( sid, 0x00, sizeof(sid) );
   memset( lta, 0x00, sizeof(lta) );

   //Connect to Host
   result = ConnectToHostGetSid( &handle, fn, sid);
   if ( result != 0 )
   {
       printf("%s Error [%d] on connect.\n", fname, result );
       return 1;
   }
```

```
   printf("%s is the session ID or LNIATA obtained from connect to
host\n",sid);

   //Get LNIATA
   GetSid(handle,lta);
   printf("%s is the session ID or LNIATA obtained from get sid\n",lta);
   quit = -1;
   firsttime = 0;

   //Sign In
   memset( request, 0x00, sizeof(request) );
   strcpy(request, "*S"); //insert your ID
   OutDataLen = strlen(request);

   result = SendData( handle,
                      State,
                      request,
                      &OutDataLen );
   if ( result != 0 )
   {
       printf("%s Error [%d] on send.\n", fname, result );
       return 1;
   }

   result = ReceiveData( handle,
                      &InData,
                      &InDataLen );
   if ( result != 0 )
   {
       printf("%s Error [%d] on receive.\n", fname, result );
       return 1;
   }

   memcpy( prtData, InData, InDataLen );
   printf( "%s\n", prtData );

   //Disconnect
   result = DisconnectFromHost( handle );
   if ( result != 0 )
   {
       printf("%s Error [%d] on disconnect.\n", fname, result );
       return 1;
   }

   return 0;
}
```

C Example 2

The following is a simple C console based application illustrating the use of the C Interface CCSAPI to sign in and communicate with Sabre:

```
#include <time.h>
#include <sys/timeb.h>

#include <errno.h>
#include <sys/types.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>


#include "csapi_structs.h"
#include "csapi_defines.h"
//Author: Ben Dash

int main( int argc,char**argv)
{
   int               GDS = 0;
   char              fn[] = "SABRE";

   static char       fname[] = "CsapiSyncTest()";
   char              request[1024];

   void*             handle;
   char*             InData;
   char              prtData[32767];
   int               InDataLen, OutDataLen;
   int               result, quit, firsttime;
   char              State = ' ';
   char              sid[7];
   char              lta[7];
   int               i, loop;

   //time specific variables
   struct _timeb tstruct1, tstruct2;
   int time_total;

   loop = 100; //number of repetitions through send "1DFWLAX"
   time_total = 0;

   memset( request, 0x00, sizeof(request) );
   memset( sid, 0x00, sizeof(sid) );
   memset( lta, 0x00, sizeof(lta) );

   //Connect to Host
   result = ConnectToHostGetSid( &handle, fn,sid);
   if ( result )
   {
       printf("%s Error [%d] on connect..\n", fname, result );
       //exit(1);
       return 1;
   }
   printf("%s is the sid obtained from connect to host \n",sid);
```

```
//Get LNIATA
GetSid(handle,lta);
printf("%s is the session id \n",lta);
quit = -1;
firsttime = 0;

//Sign In
memset( request, 0x00, sizeof(request) );
strcpy(request, "SI123456"); //insert your ID
OutDataLen = strlen(request);

result = SendData( handle,
                   State,
                   request,
                   &OutDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on send.\n", fname, result );
    return 1;
}

result = ReceiveData( handle,
                   &InData,
                   &InDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on receive.\n", fname, result );
    return 1;
}

memset( prtData, 0x00, sizeof(prtData) );
memcpy( prtData, InData, InDataLen );
printf( "%s",prtData );

result = ReceiveData( handle,
                   &InData,
                   &InDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on receive.\n", fname, result );
    return 1;
}

memset( prtData, 0x00, sizeof(prtData) );
memcpy( prtData, InData, InDataLen );
printf( "%s",prtData );

memset( request, 0x00, sizeof(request) );
//insert your ID and Password
strcpy(request, "AG<PASSWD..><123456><AA...><.><A><........>");
OutDataLen = strlen(request);

result = SendData( handle,
                   State,
                   request,
                   &OutDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on send.\n", fname, result );
```

```
    return 1;
}

result = ReceiveData( handle,
                      &InData,
                      &InDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on receive.\n", fname, result );
    return 1;
}

memset( prtData, 0x00, sizeof(prtData) );
memcpy( prtData, InData, InDataLen );
printf( "%s",prtData );

result = ReceiveData( handle,
                      &InData,
                      &InDataLen );
if ( result != 0 )
{
    printf("%s Error [%d] on receive.\n", fname, result );
    return 1;
}

memset( prtData, 0x00, sizeof(prtData) );
memcpy( prtData, InData, InDataLen );
printf( "%s",prtData );

for(i = 0; i<loop; i++)
{
   //Send "1DFWLAX"
   memset( request, 0x00, sizeof(request) );
   strcpy(request, "1DFWLAX");
   OutDataLen = strlen(request);

   //Get the current time
   _ftime( &tstruct1 );

   result = SendData( handle,
                      State,
                      request,
                      &OutDataLen );
   if ( result != 0 )
   {
      printf("%s Error [%d] on send.\n", fname, result );
      return 1;
   }

   result = ReceiveData( handle,
                      &InData,
                      &InDataLen );
   if ( result != 0 )
   {
      printf("%s Error [%d] on receive.\n", fname, result );
      return 1;
   }

   //Get the current time
   _ftime( &tstruct2 );
```

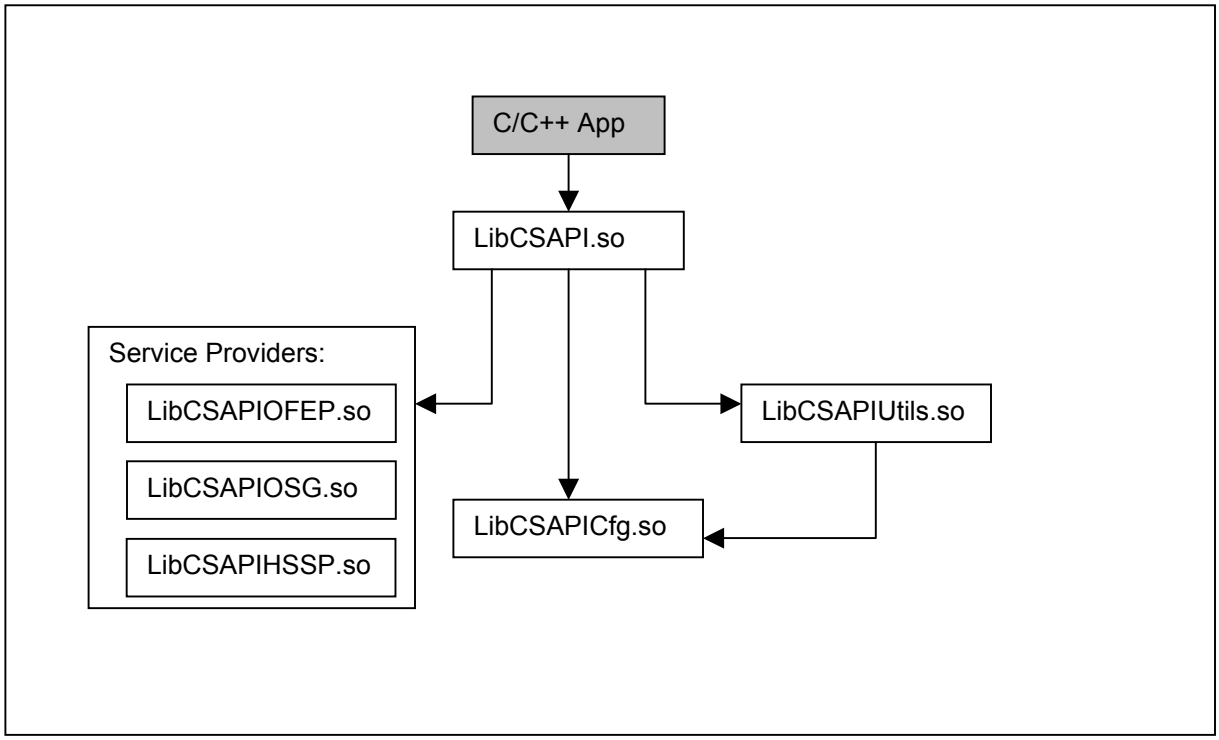
```
      //Sum total time
      time_total += ((tstruct2.time -
tstruct1.time)*1000)+(tstruct2.millitm - tstruct1.millitm);

      memset( prtData, 0x00, sizeof(prtData) );
      memcpy( prtData, InData, InDataLen );
      printf( "%s",prtData );
   }

   //Sign Out
   memset( request, 0x00, sizeof(request) );
   strcpy(request, "SO");
   OutDataLen = strlen(request);

   result = SendData( handle,
                      State,
                      request,
                      &OutDataLen );
   if ( result != 0 )
   {
       printf("%s Error [%d] on send.\n", fname, result );
       return 1;
   }

   result = ReceiveData( handle,
                      &InData,
                      &InDataLen );
   if ( result != 0 )
   {
       printf("%s Error [%d] on receive.\n", fname, result );
       return 1;
   }

   memset( prtData, 0x00, sizeof(prtData) );
   memcpy( prtData, InData, InDataLen );
   printf( "%s",prtData );

   //Disconnect
   result = DisconnectFromHost( handle );
   if ( result != 0 )
   {
       printf("%s Error [%d] on disconnect.\n", fname, result );
       return 1;
   }

 printf("Test ran for %d iterations.\nTurnaround mean time was %dMS per
iteration\n", loop, time_total/loop);

   return 0;
}
```

UNIX CCSAPI Distribution

The UNIX distribution of CCSAPI provides a C interface for a customer to write an application that communicates with Sabre, as described above.

Architecture

The diagram below illustrates the architecture of the UNIX CCSAPI:



Installation

Before using the CCSAPI it is necessary to install and configure it. Installation is basically just a matter of copying the various files provided to the appropriate place on the target machine. In the case of UNIX distributions installation involves copying the shared object, or library, files (libcsapi, libcsapiutils, libcsapicfg, libcsapiosg, libcsapiofep) to the target machine's hard disk. The location on the disk would be either the local test application directory, the directory specified in the LD_LIBRARY_PATH environment variable or one of the directories in the "/etc/ld.so.conf" file. Ensuring that the dependencies are mapped correctly using ldd.

Configuration File Location

On UNIX platforms the absolute file name may be recorded in CSAPI_CFG_FILE environment variable. If this variable is not set then the API will look for a file called "csapi.cfg" in the current working directory. If the "csapi.cfg" file is not found in either location an error will be generated and the CCSAPI will not be able to connect to Sabre.

CCSAPI Version

In order to find which version of CCSAPI you are using use the UNIX ident command. For information as to how to use the ident command refer to the UNIX man pages.

Distribution File List

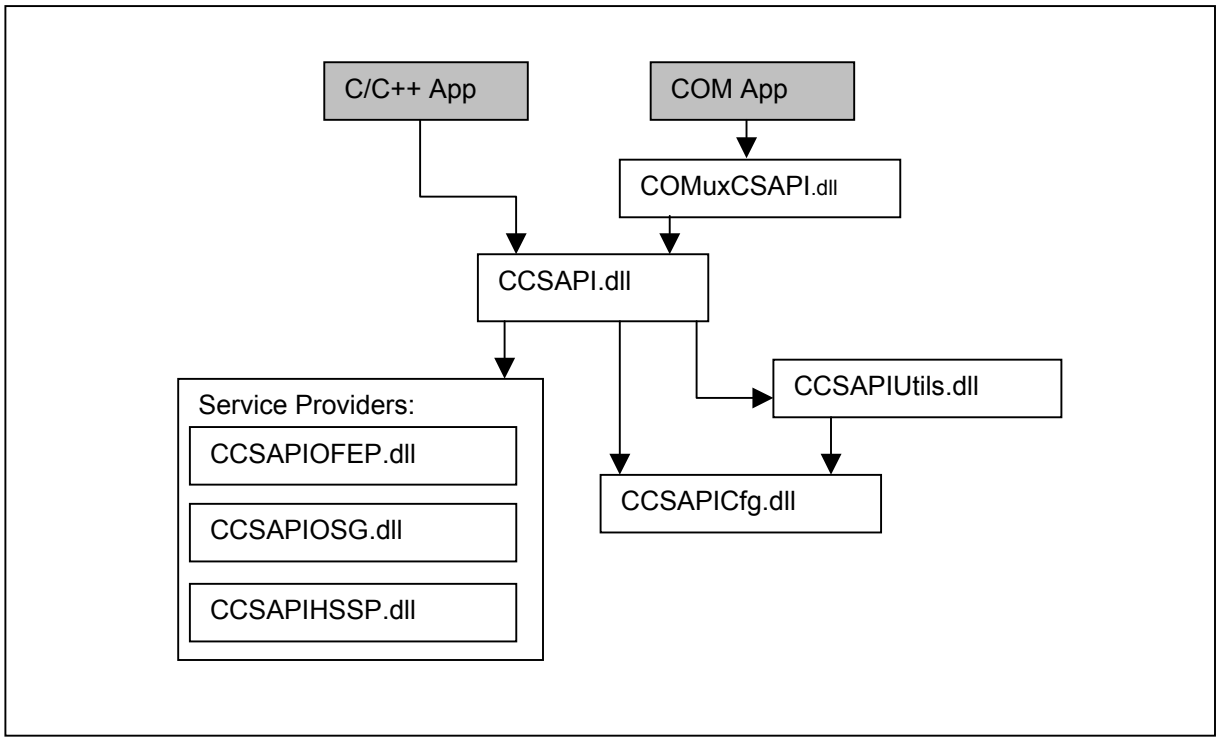
libcsapi.so
libcsapicfg.so
libcsapihssp.so
libcsapiofep.so
libcsapiosg.so
libcsapiutls.so
csapi_defines.h
csapi_structs.h
Normal.map
Travelocity.map
EWM.map
Webres.map
WebresAbacus.map
csapi.cfg
ccsapi.doc
release.txt

Win32 CCSAPI Distribution

The Win32 distribution of CCSAPI provides a C interface for a customer to write an application that communicates with Sabre, as described above. The Win32 distribution of CCSAPI also provides an ActiveX/COM interface that allows a customer to write an application using ActiveX/COM technologies that communicates with Sabre. The ActiveX/COM interface to CCSAPI is only available for Win32 and is described below.

Architecture

The diagram below illustrates the architecture of the Win32 CCSAPI (basically the architecture is identical between UNIX and Win32 except that the ActiveX/COM Interface is currently only available on Win32 platforms and the names of the Win32 DLLs are different from the names of the UNIX shared libraries/objects):



Installation

In the case of Win32 distributions installation involves copying the ".dll" files (ccsapi.dll, ccsapiutils.dll, ccsapicfg.dll, ccsapiosg.dll and ccsapiofep.dll), on Win32 distributions, to either the local test application exe folder or, preferably, the windows system directory. Once this has been done the next step for Win32 distributions involves copying the COM DLL from the installation media to the client's local hard drive, and then registering the COM component on their machine.

Copy the file "COMuxCSAPI.dll" from the installation media and place it on the client's local hard drive in the windows system directory usually "/Windows/System/" on Windows 95 machines or "/WinNT/System32/" folder on Windows NT4 machines, or in the same folder that the other CCSAPI DLLs are in.

In order to register the COM "COMuxCSAPI.dll" on the client's machine, click on the Windows "Start" button at the bottom right hand corner of the screen and select "Run". Type the following into the text box on Windows NT (assuming that the OS is installed on the C drive and the DLLs are in the "/WinNT/System32/" folder):

```
regsvr32 "C:/WinNT/System32/COMuxCSAPI.dll"
```

Then press the "OK" button and a window should appear indicating that the registration process was a success. If an error occurs at this stage check that all the DLLs are in the folder that you specified or that all the other CSAPI DLLs are accessible from that folder.

Configuration File Location

On Win32 platforms the CCSAPI will take its configuration settings from a file called "csapi.cfg" located in the Windows Directory. If the "csapi.cfg" file is not found in that directory, based on the path returned by the GetWindowsDirectory() Win32 function call, then the API will take its configuration settings from the "csapi.cfg" file located in the current working directory. If the "csapi.cfg" file is not found in either location, then an error will be generated and the CCSAPI will not be able to connect to Sabre.

CCSAPI Version

The current version of CCSAPI on the Win32 distribution can be found by running the Explorer program, browsing the folder where the CCSAPI is located, right clicking on each DLL, selecting "Properties", then selecting the "Version" tab.

Distribution File List

CCSAPI.dll
CCSAPI.lib
CCSAPIUTLS.dll
CCSAPIUTLS.lib
CCSAPICFG.dll
CCSAPICFG.lib
CCSAPIOSG.dll
CCSAPIOFEP.dll
CCSAPIHSSP.dll
ReadMe.txt
Release.txt
csapi.cfg

ccsapi.doc
csapi_structs.h
csapi_defines.h
COMuxCSAPI.h
COMuxCSAPI_i.c
CCOMTesterDlg.cpp
CCOMTesterDlg.h
Main.c
COMuxCSAPItest.asp
VBCOMTester.frm

CCSAPI COM Interface

Note that the COM interface to CCSAPI is only available on the Win32 platform. The object methods exposed on the CCSAPI ActiveX/COM COMCSAPISession interface in the COMuxCSAPI.dll are as follows:

ConnectToHostSession

Prototype: **ConnectToHostSession**(BSTR bstrRouteName)

Description: Call this method to establish a communication session with the Host.

Parameters: The bstrRouteName parameter specifies a Route from the csapi.cfg file.

Returns: To get detailed information regarding the status of the function call GetLastReturnCode.

SendDataSession

Prototype: **SendDataSession**(BSTR bstrState, BSTR bstrSendData)

Description: Call this method to send data to the Host.

Parameters: The bstrState parameter is reserved for future enhancements and is currently ignored.
The bstrSendData is the data to be sent to the Host.

Returns: To get detailed information regarding the status of the function call GetLastReturnCode.

ReceiveDataSession

Prototype: BSTR* **ReceiveDataSession**

Description: Call this method/property to get the data received in the form of a pointer to a BSTR String.

Parameters: None.

Returns: This function returns the response from Sabre in the form of a BSTR*.
To get detailed information regarding the status of the function call GetLastReturnCode.

ReceiveDataWithTimeOutSession

Prototype: BSTR* **ReceiveDataWithTimeOutSession**(SHORT sTimeOut)

Description: Call this method/property to get the data received in the form of a pointer to a BSTR String.

Parameters: The timeout parameter passed indicates a receive timeout in seconds, if the timeout expires then the GetLastReturnCode function should return 16. Using this function with a timeout of 0 is not currently permitted, it is a planned future enhancement that will make it possible to implement a polling receive call. Currently the minimum value for the timeout parameter is 1 second. However, if data arrives in less than 1 second the function call will return immediately as the data arrives. Hence this timeout would cause the function to wait for data for 1 second or less.

Returns: This function returns the response from Sabre in the form of a BSTR*. To get detailed information regarding the status of the function call GetLastReturnCode.

GetLastReturnCode

Prototype: SHORT* **GetLastReturnCode**

Description: Call this method/property to get the return code of the last function called in the form of a pointer to a short integer.

Parameters: None.

Returns: The last return code from the previously called function in the form of an int. Refer to the error code listing for more information as to the meaning of the error codes.

GetSid

Prototype: BSTR* **GetSid**

Description: Call this method/property to get the current Session ID or Line IA TA in the form of a pointer to a BSTR String.

Parameters: None.

Returns: This function returns the current Line IA TA in the form of a BSTR*. To get detailed information regarding the status of the function call GetLastReturnCode.

DisconnectFromHostSession

Prototype: **DisconnectFromHostSession**

Description: This method is called when the application needs to terminate its connection to the host and close the session.

Parameters: None.

Returns: To get detailed information regarding the status of the function call GetLastReturnCode.

COM Interface Programming Examples

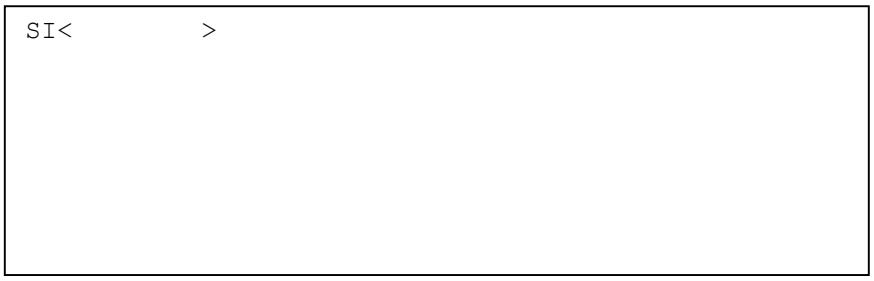
Visual Basic Example

The following is a Microsoft Visual Basic 5.0 example illustrating the use of the COM CSAPI to communicate with Sabre. The code demonstrates the Sabre sign in process:

```
Option Explicit
'Author: Ben Dash
Dim SabreTerm As Object

Private Sub Command1_Click()
    'There are 2 ways to reference the COM object
    'through VB - you can reference it through VB
    'as shown below
    Set SabreTerm = New COMCSAPISession
    'or you can reference the COM object directly
    'through its registry ProgID
    'Set SabreTerm = CreateObject("COMCSAPISession.COMCSAPISession.1")
    Dim bResult As Boolean
    Dim iLastRetCode As Integer

    bResult = SabreTerm.ConnectToHostSession("SABRE")
    iLastRetCode = SabreTerm.GetLastReturnCode()
    If iLastRetCode = 0 Then
        Text1.Text = "Connected"
        Label1.Caption = "Status: Connected"
    End If

End Sub

Private Sub Command2_Click()
    Dim bResult As Boolean
    Dim iLastRetCode As Integer

    'Send the SI signin command, note: replace 123456 with your ID
    bResult = SabreTerm.SendDataSession("NoState", "SI123456")
    iLastRetCode = SabreTerm.GetLastReturnCode()

    'Get both responses, note if response begins with AG or BA
    Text1.Text = SabreTerm.ReceiveDataSession()
    iLastRetCode = SabreTerm.GetLastReturnCode()

    Text1.Text = SabreTerm.ReceiveDataSession()
    iLastRetCode = SabreTerm.GetLastReturnCode()

    'Send the signin mask command, note: replace PASSWD with your
password etc
    'Note that AG or BA should begin the command depending
    'upon the response of the "SI" command
    bResult = SabreTerm.SendDataSession("NoState",
"AG<PASSWD..><123456><AA...><.><A><........>")
    iLastRetCode = SabreTerm.GetLastReturnCode()

    'Get both responses
    Text1.Text = SabreTerm.ReceiveDataSession()
```

```
    iLastRetCode = SabreTerm.GetLastReturnCode()

    Text1.Text = SabreTerm.ReceiveDataSession()
    iLastRetCode = SabreTerm.GetLastReturnCode()
    'if your id, password and the rest of your signin mask
    'were filled in correctly you should now be signed into Sabre

End Sub

Private Sub Command3_Click()
    Dim bResult As Boolean
    Dim iLastRetCode As Integer

    bResult = SabreTerm.DisconnectFromHostSession()
    iLastRetCode = SabreTerm.GetLastReturnCode()
    Text1.Text = "Disconnected"
    Label1.Caption = "Status: Not Connected"
End Sub
```

This code will result in the following 4 responses from Sabre:


SI< >
AGENT SIGN IN
CURRENT PASSCODE _ ID <123456> SUF <AA >
DUTY CODE <.> AREA <A> NEW PASSCODE <……..>
SI< >
AGENT SIGN IN
CURRENT PASSCODE _ ID <123456> SUF <AA >
DUTY CODE <.> AREA <A> NEW PASSCODE <……..>
ENTER CORRECT PASSCODE

Active Server Page Example

The following is an Active Server Page example illustrating the use of the COM CCSAPI to communicate with Sabre:

```
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1">
   <meta name="Author" content="Ben Dash">
   <meta name="GENERATOR" content="Mozilla/4.5 [en] (WinNT; I)
[Netscape]">
   <title>COMuxCSAPI Test</title>
</head>
<body text="#000000" bgcolor="#FFFFCC" link="#999900" vlink="#006600"
alink="#FF0000">

<%
Count = 1
Application("Counter7") = Count

Set Session("test9") =
Server.CreateObject("COMCSAPISession.COMCSAPISession.1")
Response.Write "Server.CreateObject() Called <br>"

CStatus = Session("test9").ConnectToHostSession("SABRE")
Response.Write "ConnectToHostSession(SABRE) Called <br>"

Error = Session("test9").GetLastReturnCode
if not Error = 0 Then
               Response.Write "Error: " & Error & "<br>"
end if

CStatus = Session("test9").SendDataSession("NoState", "¥J")
Response.Write "SendDataSession(NoState, ¥J) Called <br>"

Error = Session("test9").GetLastReturnCode
if not Error = 0 Then
               Response.Write "Error: " & Error & "<br>"
end if

Data = Session("test9").ReceiveDataSession
Response.Write "ReceiveDataSession() Called <br>"

Response.Write "Data: " & Data & "<br>"

CStatus = Session("test9").DisconnectFromHostSession()
Response.Write "DisconnectFromHostSession() Called <br>"

Error = Session("test9").GetLastReturnCode
if not Error = 0 Then
               Response.Write "Error: " & Error & "<br>"
end if
%>

</body>
</html>
```

C++ Example

The following is a simple C++ console based application illustrating the use of the COM CCSAPI to communicate with Sabre:

```
#include <stdio.h>

#include "stdafx.h"

#include "COMuxCSAPI.h"
#include "COMuxCSAPI_i.c"
//Author: Ben Dash

int main(void)
{
   USES_CONVERSION;

   HRESULT hr;
   ICOMCSAPISession *pCOMuxCSAPIOb, *pCOMuxCSAPIOb2;
   short sReturnCode = 0;
   BSTR bstrResultHolder;
   char acData[1024];


   hr = CoInitialize(NULL);

   if(FAILED(hr))
   {
      printf("CoInitialize() Failed\n");
      return 0;
   }

   hr = CoCreateInstance(CLSID_COMCSAPISession,
      NULL,
      CLSCTX_INPROC_SERVER,
      IID_ICOMCSAPISession,
      (void**) &pCOMuxCSAPIOb);

   if(FAILED(hr))
   {
      printf("CoCreateInstance() Failed\n");
      return 0;
   }

   pCOMuxCSAPIOb->ConnectToHostSession(L"SABRE");

   pCOMuxCSAPIOb->GetLastReturnCode(&sReturnCode);

   if(sReturnCode == 0)
   {
      printf("Connected...\n");
   }
   else
   {
      printf("Connect Failed...\n");
      return 0;
   }

   pCOMuxCSAPIOb->SendDataSession(L"NoState", L"¥J");
```

```
   pCOMuxCSAPIOb->ReceiveDataSession(&bstrResultHolder);

   wprintf(L"Returned Data through wprintf:\n%s\n", bstrResultHolder);

   wsprintf(acData,"%S",bstrResultHolder);
   printf("Returned Data through printf after wsprintf:\n%s\n", acData);

   pCOMuxCSAPIOb->DisconnectFromHostSession();

   // /*

   hr = CoCreateInstance(CLSID_COMCSAPISession,
      NULL,
      CLSCTX_INPROC_SERVER,
      IID_ICOMCSAPISession,
      (void**) &pCOMuxCSAPIOb2);

   if(FAILED(hr))
   {
      printf("CoCreateInstance() Failed\n");
      return 0;
   }

   pCOMuxCSAPIOb2->ConnectToHostSession(L"SABRE");

   pCOMuxCSAPIOb2->GetLastReturnCode(&sReturnCode);

   if(sReturnCode == 0)
   {
      printf("Connected...\n");
   }
   else
   {
      printf("Connect Failed...\n");
      return 0;
   }

   pCOMuxCSAPIOb2->SendDataSession(L"NoState", L"¥J");

   pCOMuxCSAPIOb2->ReceiveDataSession(&bstrResultHolder);

   wprintf(L"Returned Data through wprintf:\n%s\n", bstrResultHolder);

   pCOMuxCSAPIOb->DisconnectFromHostSession();

   // */

   CoUninitialize();

   return 0;
}
```

Frequently Asked Questions

Where Do I Go For Help?

For assistance with the CCSAPI contact your Sabre Account Representative or eCommerce Development Manager as appropriate for your case. They will be able to determine the type of assistance that you require and will be able to forward your enquiry to the appropriate Sabre support team.

Why Do I Get a Connect Error?

Connect errors are usually caused by a misconfiguration of the csapi.cfg file. Before contacting Sabre for Support please ensure that you have read and followed the instructions on installing you csapi.cfg file and also have configured your csapi.cfg file correctly. Also ensure that you have Sabre connectivity to your development machine though the methods mentioned in the documentation.

How Do I Signin To Sabre?

There are 2 methods to signin to Sabre. Your LNIATA must be configured specifically for one of these methods.

The long signin consists of sending “SI”, followed by your Sabre ID number, followed by an End Item Sabre Character, followed by your Sabre password. Then calling the appropriate receive function will confirm or deny your signed in status.

The short signin consists of sending “SI”, followed by your Sabre ID number. Then you call the appropriate receive function 2 times. The second receive buffer contains what is known as a Sabre “mask”. You need to take the first 2 characters of this mask, “AG” in the case of “AGENT SIGN IN” masks, and then append all the protected mode mask fields. The protected mode mask fields are the fields surronded in “<” and “>”. You send this entry, which usually resembles “AG<PASSWORD><123456>…”, to Sabre. You will then receive 2 responses that will confirm or deny your signed in status.

Why Does Receive Give Error 16?

Error 16 is a timeout error. All Sabre entries take different time to be processed by Sabre. If your entry returns too slowly for your receive timeout then you will get an error 16 from receive. To prevent this increase the receive timeout in the csapi.cfg file or use the specified timeout version of receive (ReceiveDataWithTimeout).