GATE Release 3.5

VALUES API Member Front End Development Guide

Volume 1 - Call Interface

© Eurex 2006

Deutsche Börse AG (DBAG), Eurex Frankfurt AG, Eurex Bonds GmbH (Eurex Bonds) and Eurex Repo GmbH (Eurex Repo) are corporate entities and are registered under German law. Eurex Zürich AG is a public company and is registered under Swiss law. The administrating and operating institutions of Eurex Deutschland and Eurex Zürich (Eurex Exchanges) are Eurex Frankfurt AG (Eurex) and Eurex Zürich AG (Eurex), respectively. All intellectual property, proprietary and other rights and interests in this publication and the subject matter hereof (other than certain trademarks and service marks listed below) are owned by DBAG and its affiliates and subsidiaries including, without limitation, all patent, registered design, copyright, trademark and service mark rights. While reasonable care has been taken in the preparation of this publication to provide details that are accurate and not misleading at the time of publication DBAG, Eurex, the Eurex Exchanges and their respective servants and agents (a) do not make any representations or warranties regarding the information contained herein, whether express or implied, including without limitation any implied warranty of merchantability or fitness for a particular purpose or any warranty with respect to the accuracy, correctness, quality, completeness or timeliness of such information, and (b) shall not be responsible or liable for any third party's use of any information contained herein under any circumstances, including, without limitation, in connection with actual trading or otherwise or for any errors or omissions occurring in this publication.

This publication is published for information only and shall not constitute investment advice. This brochure is not intended for solicitation purposes but only for use as general information. All descriptions, examples and calculations contained in this publication are for illustrative purposes only.

Eurex offers services directly to members of the Eurex Exchanges. Those who desire to trade any products available on the Eurex Exchanges or who desire to offer and sell any such products to others should consider legal and regulatory requirements of those jurisdictions relevant to them, as well as the risks associated with such products, before doing so.

Eurex derivatives (other than the DAX® Futures contract, Dow Jones STOXX 50 Futures contract, Dow Jones EURO STOXX 50 Futures contract, Dow Jones STOXX 600 Banking Sector Futures contract, Dow Jones EURO STOXX Banking Sector Futures contract, Dow Jones Global Titans 50 Futures contract and Eurex interest rate derivatives) are currently not available for offer, sale or trading in the United States or by United States persons.

Trademarks and Service Marks

Buxl®, DAX®, Eurex®, Eurex Bonds®, Eurex Repo®, Eurex US®, iNAV®, MDAX®, SDAX®, StatistiX®, TecDAX®, Xetra® and XTF Exchange Traded Funds® are registered trademarks of Deutsche Börse AG.

SMI® is a registered trademark of SWX Swiss Exchange. STOXXSM and Dow Jones EURO STOXX/STOXXSM 600 Sector Indexes as well as the Dow Jones EURO STOXXSM 50 Index and the Dow Jones STOXXSM 50 Index are service marks of STOXX Ltd. and/or Dow Jones & Company, Inc. Dow Jones and Dow Jones Global Titans 50SM Index are service marks of Dow Jones & Company, Inc. The derivatives based on these indexes are not sponsored, endorsed, sold or promoted by STOXX Ltd. or Dow Jones & Company, Inc., and neither party makes any representation regarding the advisability of trading or of investing in such products.

The names of other companies and third party products may be the trademarks or service marks of their respective owners.

Table of Contents

1 Introduction 6
1.1 VALUES API 6
2 VALUES API Call Interface Concepts 8
2.1 Overview 8
2.2 Session Management Services 11
2.2.1 Overview 12
2.2.2 Initiating a VALUES Session 12
2.2.3 Terminating a VALUES Session Normally 15
2.2.4 Terminating a VALUES Session Abnormally 16
2.3 Security Management Services 16
2.3.1 Overview 16
2.3.2 Logging Into an Exchange Application 17
2.3.3 Receiving a Login Response 19
2.3.4 Logging Out from an Exchange Application Normally 20
2.3.5 Receiving a Logout Response 22
2.3.6 Logging Out from an Exchange Application Abnormally 23
2.4 Request Management Services 24
2.4.1 Overview 24
2.4.2 Submitting an Application Request 24
2.4.3 Receiving an Application Response 26
2.5 Subscription Management Services 28
2.5.1 Overview 28
2.5.2 Broadcast Extension 29
2.5.3 Identifying Available Data Streams 30
2.5.4 Subscribing to a Data Stream 30
2.5.5 Receiving Subscription Responses 33
2.5.6 Receiving Subscription Data 35
2.5.7 Unsubscribing from a Data Stream Normally 36
2.5.8 Receiving Unsubscription Responses 38
2.5.9 Unsubscribing from a Data Stream Abnormally 39
2.6 Integrating VALUES Events 40
2.7 Recovery Management Services 42
2.8 Multi-User Capability 43
2.9 Xervices, Xervice Classes and Multi-Exchange Capability 44
2.10 VALUES API Backwards Compatibility Concepts 44
3 VALUES API Call Interface Reference 46
3.1 Overview 46
3.2 State Diagrams 48
3.2.1 Overview 48
3.2.2 Normal Operation 48
3.2.3 Exception Handling 52

3.3 VCI_Connect 56
3.3.1 Overview 56
3.3.2 VALUES Call Interface Version 59
3.3.3 The Connect Application Callback 59
3.4 VCI_Disconnect 60
3.5 VCI_Dispatch 61
3.6 VCI_Login 63
3.6.1 The Login Application Callback 65
3.7 VCI_Logout 66
3.8 VCI_Submit 68
3.8.1 The Submit Application Callback 70
3.9 VCI_Subscribe 70
3.9.1 The Subscription Application Callback 73
3.10 VCI_Unsubscribe 73
3.11 Application Callback Function Type 76
4 VALUES API Usage Examples 83
4.1 Overview 83
4.2 Initiating a VALUES Connection 83
4.3 Application Dispatch upon Event Notification 86
4.4 Receiving Connection Events 87
4.5 Logging on to Exchange Services 89
4.6 Receiving Login Responses 92
4.7 Submitting Application Requests 93
4.8 Receiving Application Responses 95
4.9 Subscribing to a Data Stream 96
4.10 Receiving Subscription Data 98
4.11 Unsubscribing from a Data Stream 100
4.12 Logging off from an Exchange Service 100
4.13 Terminating a VALUES Session 101
4.14 Auxiliary Functions of an End User Application 102
5 VALUES API Completion Codes (Call Interface) 103
6 VALUES API Call Interface Field Descriptions 105
6.1 Overview 105
6.1.1 Field Characteristics 105
6.1.2 Initialization Guideline 106
6.1.3 Template for the Call Interface Field Descriptions 106
6.2 Call Interface Field Descriptions 106
6.2.1 appDescr (ReqCntrlT) 106
6.2.2 applClass (XerviceInfoT) 107
6.2.3 applPrevVersion (CallBkAppDataT) 107
6.2.4 applVersion (LoginReqDataT, SubsReqDataT, CallBkAppDataT, XerviceInfoT) 107
6.2.5 appReq (SubmitReqDataT) 108
6.2.6 appReqBlockSize (CallBkAppDataT, SubmitReqDataT) 108

6.2.7 appReqData (CallBkAppDataT) 109
6.2.8 appRespBlockSize (CallBkAppDataT) 109
6.2.9 appRespData (CallBkAppDataT) 109
6.2.10 authorizationData (LoginReqDataT, SubsReqDataT) 110
6.2.11 authorizationDataLength (LoginReqDataT, SubsReqDataT) 110
6.2.12 brcSubject (CallBkAppDataT) 111
6.2.13 closure (LoginReqDataT) 111
6.2.14 complCode (statusDataT) 111
6.2.15 complSeverity (statusDataT) 112
6.2.16 complText (statusDataT) 112
6.2.17 connectionID (ReqCntrlT, CnctRespDataT) 113
6.2.18 custBlockSize (AppCntxtDataT) 113
6.2.19 custData (AppCntxtDataT) 114
6.2.20 dbApplID (ReqCntrlT) 114
6.2.21 exchApplId (XerviceInfoT) 115
6.2.22 exchDscrName (XerviceInfoT) 115
6.2.23 funcResult (LoginRespDataT) 115
6.2.24 loginID (ReqCntrlT, LoginRespDataT) 116
6.2.25 password (CnctReqDataT) 116
6.2.26 prodMode (CnctRespDataT) 117
6.2.27 reqID (ReqCntrlT) 117
6.2.28 resubmitFlag (ReqCntrlT) 118
6.2.29 resubmitNo (ReqCntrlT) 118
6.2.30 streamType (CallbkAppDataT, SubsReqDataT) 119
6.2.31 subject (CallBkAppDataT) 119
6.2.32 subject (SubsRegDataT) 119
6.2.33 subjectLength (SubsReqDataT) 120
6.2.34 subsID (CallBkAppDataT, SubsRespDataT) 120
6.2.35 subsSubject (SubsRegDataT) 121
6.2.36 techComplCode (StatusDataT) 121
6.2.37 techComplSeverity (StatusDataT) 121
6.2.38 techComplText (StatusDataT) 122
6.2.39 userID (CnctReqDataT) 122
6.2.40 userID (LoginReqDataT) 123
6.2.41 userID (SubsReqDataT) 123
6.2.42 VCIver (ReqCntrlT) 123
6.2.43 VMQname (CnctRespDataT) 124
7 Data Definitions 125
8 Glossary 126

1 Introduction

The VALUES API (Virtual Access Link Using Exchange Services Application Programming Interface) is an interface which provides the functionality and flexibility needed to serve as a standard open interface to Exchange services.

The VALUES API interface specification is the developer's guide for designing and implementing applications which use the VALUES API standard open interface to Exchange services. The Member Front End Development Guide is the final version of the VALUES API interface specification and describes the structure of the interface, how to use the interface, and the associated development environment requirements.

1.1 VALUES API

The VALUES API provides member systems with the single point of access to Exchange services. The VALUES API can be utilized for both, human driven and computer driven trading activities. *Figure 1.1* shows how the VALUES API fits into the overall Exchange infrastructure.

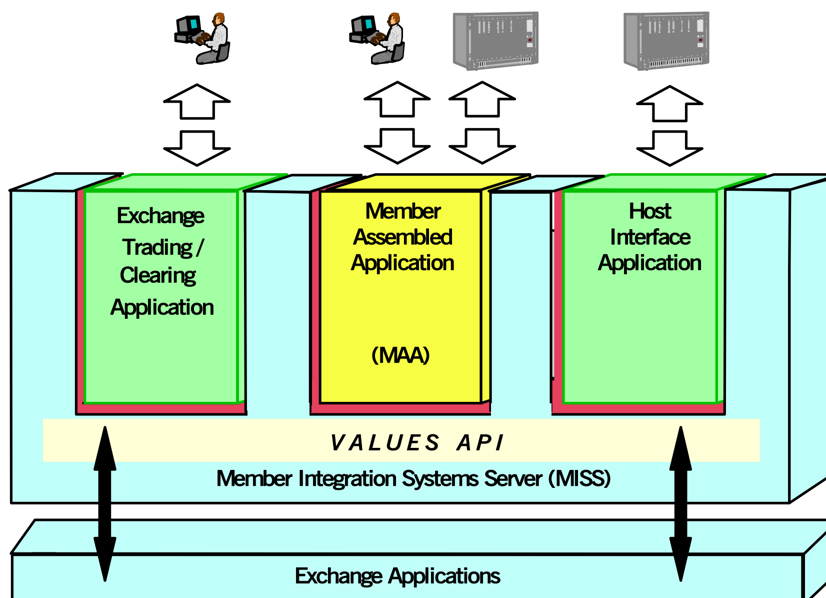


Figure 1.1 - The VALUES API in the Context of Exchange Infrastructure

In *Figure 1.1* and throughout the interface specification, Exchange applications refer to any Exchange provided application which delivers trading services (e.g., submission of an order, inquiry for news list), trading support, and clearing services to an end user application. End user applications refer to any application which receives services via the VALUES API. Furthermore, a user for the interface specification is defined as any member, trader, or participant who receives Exchange services via the VALUES API.

The VALUES API is composed of a set of technical entry points known as the VALUES API Call Interface and functional entry points known as VALUES API application requests:

- **VALUES API Call Interface**
  The VALUES API Call Interface is a set of C functions called by an application to access Exchange services.
- **VALUES API Application Requests**
  The VALUES API application requests provide the functional layout and format related information required to access Exchange services and are used in conjunction with the Call Interface entry points as depicted in *Figure 2.1*.

The split of the VALUES API into technical and functional components minimizes the impacts of newly released application requests to developers. Developers are encouraged to maintain a clear separation of their technical framework that handles the VALUES Call Interface calls from the functional aspects like message formatting etc.

The VALUES API can be expanded to support new functionality offered in each release of Exchange applications. The expansion of the VALUES API is achieved by introducing new application requests which correspond to the new functionality of a release. The goal is to keep the Call Interface stable through successive releases of the VALUES API and the Exchange applications. In addition, the VALUES API conceptually supports backwards compatibility for subsequent releases in the future.

2 VALUES API Call Interface Concepts

In this section, the main concepts of the VALUES API Call Interface (VCI) are explained. After a brief overview, Call Interface concepts are described in more detail in the following sections, grouped by service categories.

2.1 Overview

The VALUES API Call Interface is a set of C functions used to communicate with Exchange applications. Through the VALUES API Call Interface, end user applications request Exchange services by passing application requests and receiving application responses. Additionally, broadcast data is received through the VALUES API Call Interface in response to subscription requests.

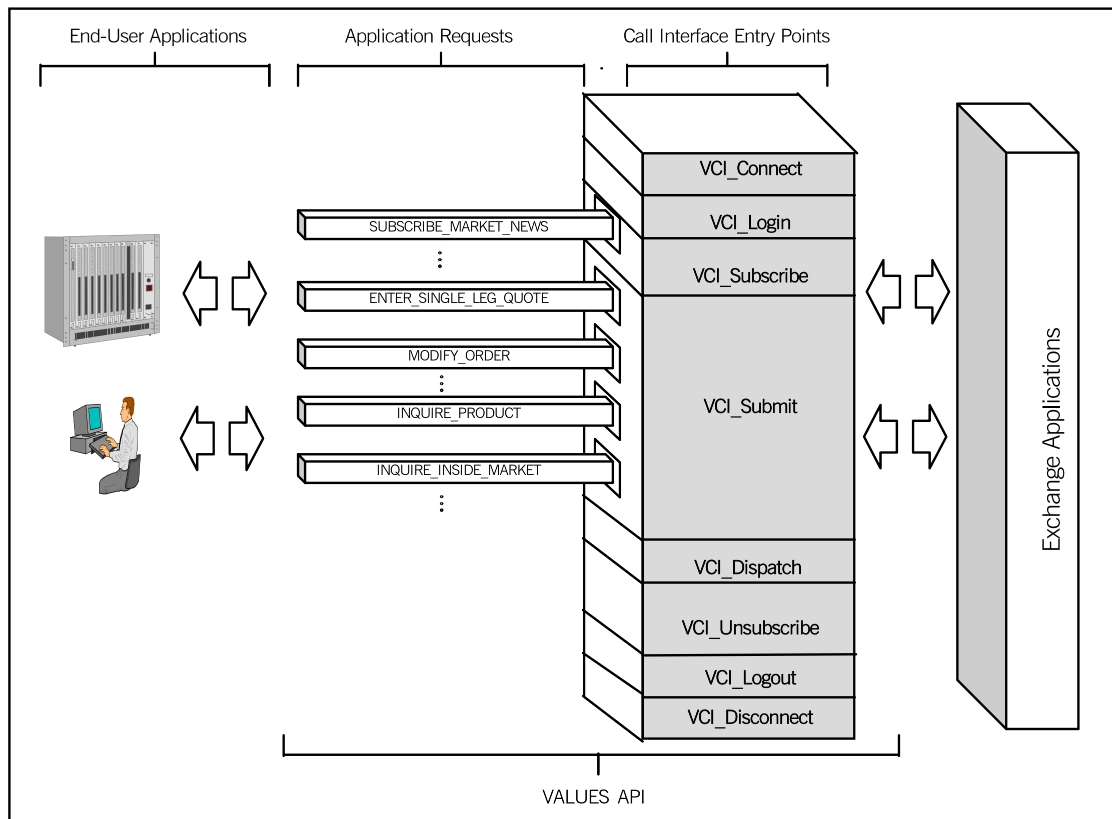


Figure 2.1 - The VALUES API Framework - Call Interface Entry Points

The Call Interface consists of a fixed number of entry points, as shown in *Figure 2.1*, which are used to establish a session, login to a specific Exchange application, transmit application requests, request broadcast data and receive responses. The following table summarizes the Call Interface entry points and how they are grouped into service categories:

Service Category	Entry Point	Purpose
Session Management	VCI_Connect	Used to establish communication between the end user application and GATE (Generic Access To Exchanges).
	VCI_Disconnect	Used to end access to GATE.
	VCI_Dispatch	Used to forward initial status and service specific information of Exchange applications, state transitions and exceptions to the application's connect callback [1].
Security Management	VCI_Login	Used to gain access to specific Exchange applications.
	VCI_Logout	Used to end access to specific Exchange applications.
	VCI_Dispatch	Used to forward Login and Logout notification responses and exceptions to the login application callback.
Subscription Management	VCI_Subscribe	Used to request access to broadcast data streams.
	VCI_Unsubscribe	Used to end access to broadcast data streams.
	VCI_Dispatch	Used to forward Subscribe and Unsubscribe responses, subscription data and exceptions to the subscribe application callback.
Request Management	VCI_Submit	Used to send application requests to Exchange applications.
	VCI_Dispatch	Used to forward Exchange application responses and exceptions to the submit application callback.

Table 2.1 - VALUES API Entry Points

1. A callback is a concept used in event-driven programming (for example GUI programming) and is an event specific function which is executed on occurrence of the corresponding event.

A hierarchical structure must be followed when using the VALUES API Call Interface. *Figure 2.2* graphically depicts the usage hierarchy.

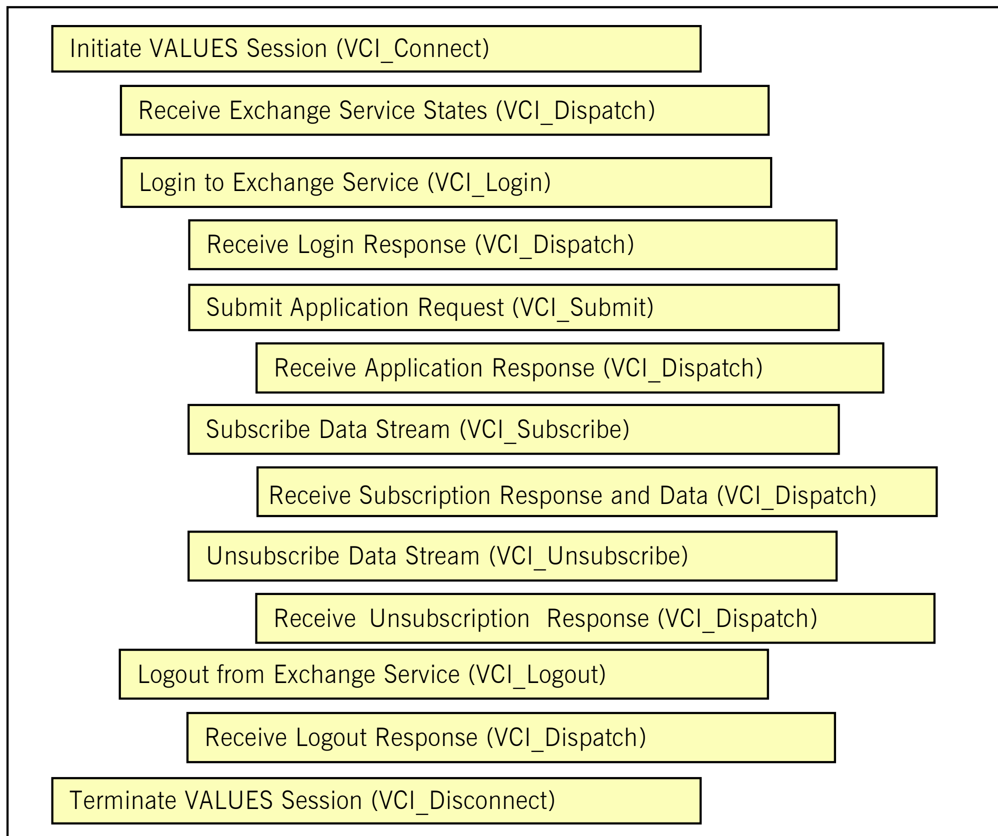


Figure 2.2 - VALUES API Usage Hierarchy

The level of right indentation depicts the level of hierarchy, and shows the related entry points at the same level of indentation. Tasks/entry points indented right depend on the tasks/entry points to the left. For example, use of VCI_Connect is a prerequisite to calling VCI_Login. Additionally, entry points usage follows a top to bottom sequence. The order in which the individual entry point must be used and the number of times they can be used is detailed in the following sections.

Note: The figure reflects the situation of a Xervice that implements Broadcast Extension. Xervices that do not have these allow broadcast subscriptions based on a VALUES session alone, i.e. without a valid login context. Please see *section 2.5.2* for details.

The VALUES API Call Interface entry points and associated application callbacks are described in the following sections.

For each service category, an explanation is given of the associated concept; e.g., what is a session and what are the key data for session management. Additionally, the information flow for each Call Interface entry point and application callback associated with each service category is explained.

The information flow for each service category is explained via a processing walkthrough. For the purpose of explaining the concepts of the VALUES API, a user interacting with a GUI application is assumed in the processing flow model. The processing walkthrough is shown graphically using the following model.

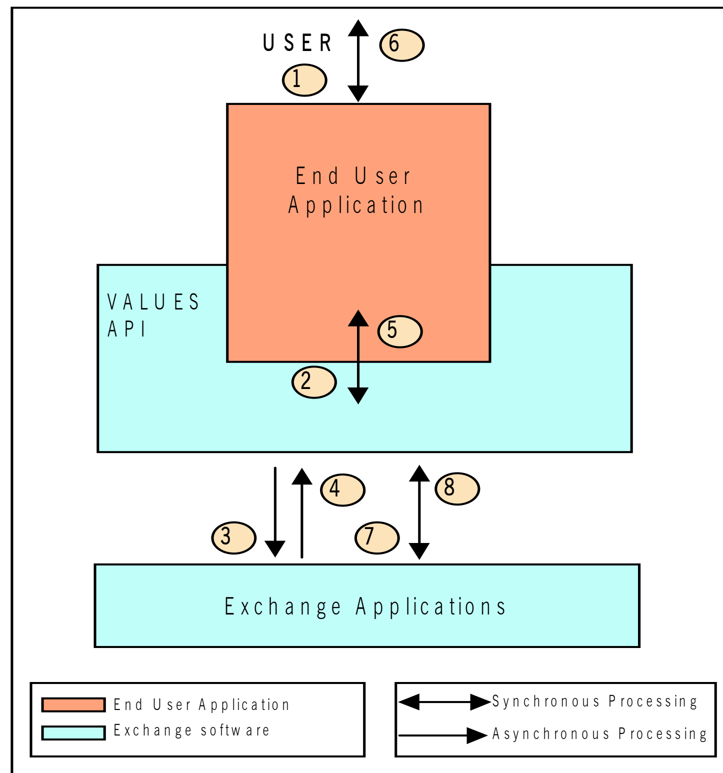


Figure 2.3 - VALUES API Information and Processing Flow Model

Each processing walkthrough is presented independently of the end user application location on either a workstation or a MISS; i.e., an end user application process may run on either a workstation or a MISS. Under normal operation, running VALUES applications on a workstation is recommended. Each processing step is assigned a number which links the sequence of processing with explanatory text in a table. A processing step next to an arrow head depicts the direction for the specific step. A processing step next to an arrow line describes both directions. This table containing the processing steps with explanatory text also describes key input and output data.

Processing is described as synchronous (double-headed arrows) and asynchronous (single-headed arrows) and is to be seen from an end user application perspective. End user application and Exchange applications code/processing are shown in different shades of gray.

2.2 Session Management Services

In this section, an overview on what a session is and what sessions are used for is given. A detailed explanation of the session startup and session shutdown is also provided. In the last section, the mechanism for integrating VCI application requests and responses with end user applications is shown.

2.2.1 Overview

A VALUES session is a control technique for managing communications between an end user application and VALUES API. All communication between end user applications and Exchange applications via GATE is built on top of a session.

To use Exchange services, an end user application must first start a VALUES session. When finished interacting with VALUES API, an end user application must end the VALUES session. Multiple end user applications or application instances may run in parallel, each with its own VALUES session. However, only one VALUES session may be established per application process.

Note: The VALUES API is not thread safe. If a multi-threaded application issues a call to a VCI entry point with any VCI call still running in another thread, the second call will fail. Reentrant use of VCI entry points is also not allowed – from inside an application callback that is invoked by VALUES, no VCI calls must be used. Offending invocations are rejected. A multi-threaded end user application is responsible for synchronizing it's threads accordingly.

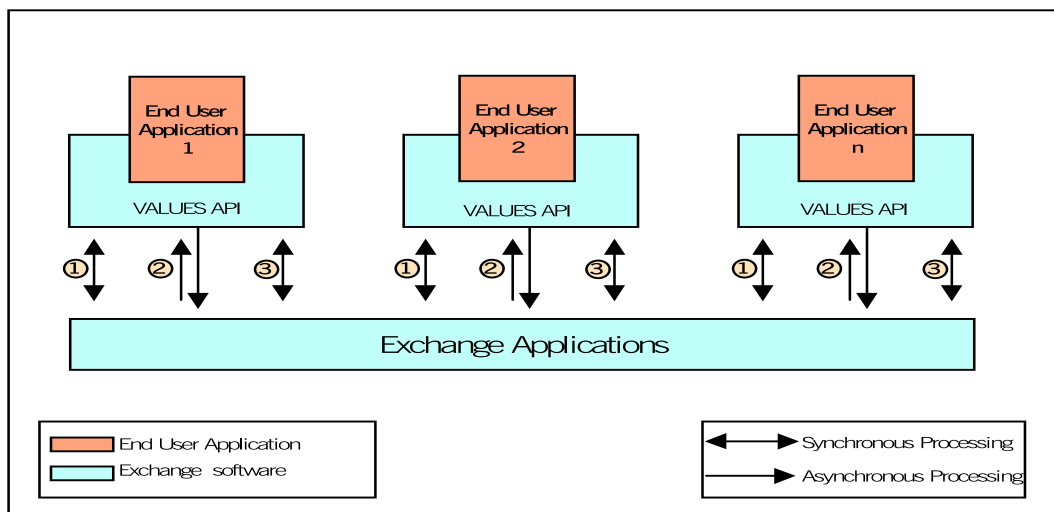


Figure 2.4 - VALUES Session Concept

Figure 2.4 above shows the execution of multiple end user applications process in parallel. Each application process initiates its own VALUES session (1), interacts with Exchange applications (2) and then terminates its session (3).

2.2.2 Initiating a VALUES Session

In this section the processing flow during the startup of a session is described. The VCI_Connect entry point is explained in the context of a user starting an end user application. VCI_Connect is a prerequisite for the use of any other VALUES API entry point. *Figure 2.5* graphically depicts the processing involved in initiating a VALUES session. The processing steps are described in *Table 2.2*.

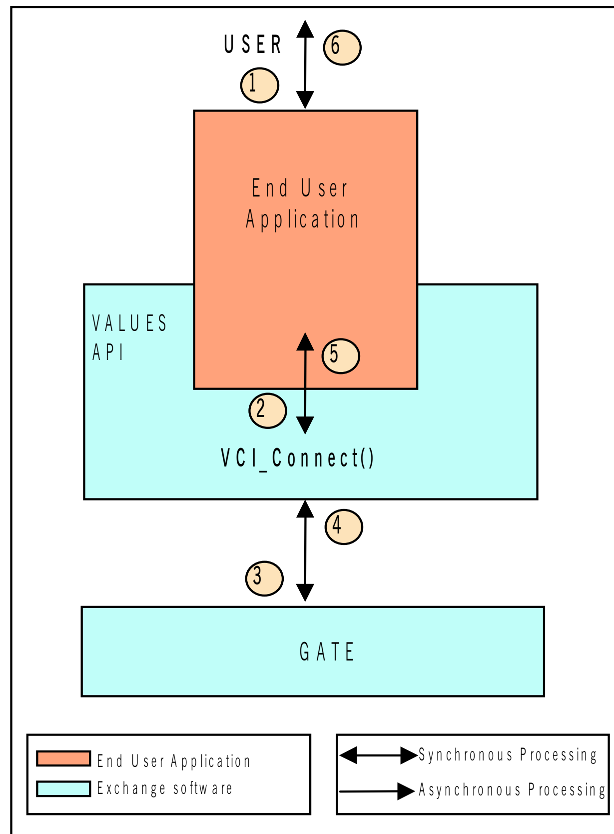


Figure 2.5 - Connecting to GATE

Step	Description	Input	Output
1	The user starts an end user application. The application initializes itself.	n/a	n/a
2	The application calls VCI_Connect to start a VALUES session. User ID (UID), password (PW) and an application callback reference are passed. The user ID and the password are authenticated using operating system security functions. The application callback reference (see *section 2.6* for information on application callbacks) is used to notify the application of information that identifies available services (see *section 2.9*), initial status and state transitions of Exchange applications and GATE after the session is established. The user ID and password enable authentication and authorization of the session request. The application process waits for VCI_Connect to complete.	UID, PW, callback reference	n/a
3	VCI_Connect forwards the session startup request to GATE and waits for the processing to be completed.	UID, PW	n/a
4	GATE authenticates the connection request and allocates a connection ID (CID). GATE returns a completion code (status) and, if the request was successful, a connection ID to VCI_Connect.	n/a	status, CID
5	VCI_Connect installs the application callback for later use. It then returns the completion code (status), the production mode (Prod Mode) and the name of the VALUES Message Queue (VMQ) to the end user application. Details on the VMQ are provided in *section 2.6*.	n/a	status, Prod Mode, VMQ-name
6	The end user application completes its initialization including establishing access to the VMQ and returns control to the user. Note: The VMQ must be monitored for events immediately after successful session establishment.	n/a	n/a

Table 2.2 - VCI_Connect Process Flow

The end user application has now established a session with GATE. At this point, one or several logins or subscriptions to an Exchange application can be initiated or the session can be terminated.

2.2.3 Terminating a VALUES Session Normally

In this section, the processing flow during the shutdown of a session is described. The VCI_Disconnect entry point is explained in the context of a user requesting termination of an end user application. The application must log out from Exchange applications and end all subscriptions before performing a VCI_Disconnect. *Figure 2.6* graphically depicts the processing involved in session termination. The processing steps are described in *Table 2.3*.

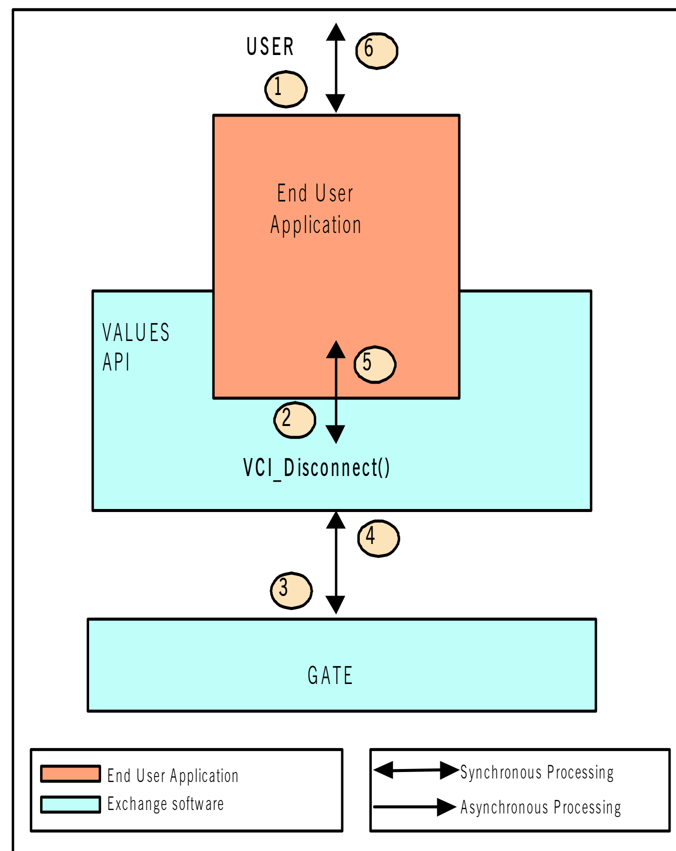


Figure 2.6 - Disconnecting from Technical Services GATE

Step	Description	Input	Output
1	The user initiates application shutdown. The application logs out from Exchange Applications and ends all subscriptions.	n/a	n/a
2	The application calls VCI_Disconnect to process the shutdown request. The application waits for VCI_Disconnect to complete.	connection ID	n/a
3	VCI_Disconnect forwards the session termination request to GATE and waits for the processing to be completed.	CID	n/a

Table 2.3 - VCI_Disconnect Process Flow

Step	Description	Input	Output
4	GATE confirms the existence of the session and the user's right to terminate it. The session is terminated and a completion code (status) is returned to VCI_Disconnect.	n/a	status
5	VCI_Disconnect cancels the application callback associated with the session after invoking it. The completion code (status) is then returned to the application.	n/a	status
6	The end user application can now inform the user of the successful disconnection and complete the shutdown processing.	n/a	n/a

Table 2.3 - VCI_Disconnect Process Flow

The end user application has now ended its VALUES session. At this point, the application can call VCI_Connect to re-establish a session with GATE.

2.2.4 Terminating a VALUES Session Abnormally

A range of exceptions can occur in case of failure of hardware or software components. If the application is unable to disconnect normally, exception handling must take place.

Exception handling is supported by VALUES API through completion status and application callback invocation. When an exception occurs, VALUES API responds with invocation of registered application callbacks in a defined sequence. Please refer to *section 3.2.3* for detailed information on exception handling.

2.3 Security Management Services

In this section, an overview of VALUES API security management services is given. The Call Interface entry points VCI_Login and VCI_Logout and their relationship to VCI_Connect are explained.

2.3.1 Overview

VALUES API secures two levels of application access; access to GATE and access to Exchange applications. Securing access to GATE and Exchange applications consists of authenticating the user and authorizing user requests. The mechanisms and associated data used by VALUES API to secure access are shown in *Table 2.4*.

Access to and use of the VALUES API Call Interface must be authenticated and authorized. Access control has to be provided for a series of layered components:

- Workstation or MISS operating system
- End user applications
- Exchange applications

The VALUES API interface specification focuses only on security required for Exchange applications and end user applications which use VALUES API. *Table 2.4* maps security mechanisms and responsi-

bilities to the different layers to be secured. User is defined in *Table 2.4* as any member, trader, or participant who receives Exchange services via the VALUES API.

Access Control	Responsible	Mechanism	Data
Workstation and MISS	User	User defined	User defined
GATE	Exchange	VCI_Connect	VALUES User ID and Password (MISS Operating System User ID/Password)
VALUES API-based applications	User	User defined	User defined
Exchange applications	Exchange	VCI_Login	Exchange applications User ID and authorization data.

Table 2.4 - VALUES API Security Mechanisms

As discussed in *section 2.2*, VCI_Connect is used to start a VALUES session. It is also the mechanism used to secure access to GATE.

Access control to the Exchange applications is supported via the VCI_Login entry point. VCI_Login must be used to obtain user authorization to each Exchange application. In the following sections, use of the VCI_Login and VCI_Logout entry points is described.

2.3.2 Logging Into an Exchange Application

In this section, the processing flow of a login request is described. The VCI_Login entry point is explained in the context of a user attempt to gain access to an Exchange application. The end user application must have previously established a VALUES session via VCI_Connect. Several logins are possible within one connection to VALUES API. Please refer to *section 2.8* for detailed information on multi-user capability.

The application has to pass a Xervice identifier (dbApplID) in order to specify the desired Exchange Service (=Xervice) for logging in. Please refer to *section 2.9* for information on how to obtain the correct value for dbApplID, and for details on the Xervice concept.

Figure 2.7 graphically depicts the processing involved in logging into an Exchange application. The processing steps are explained in *Table 2.5*.

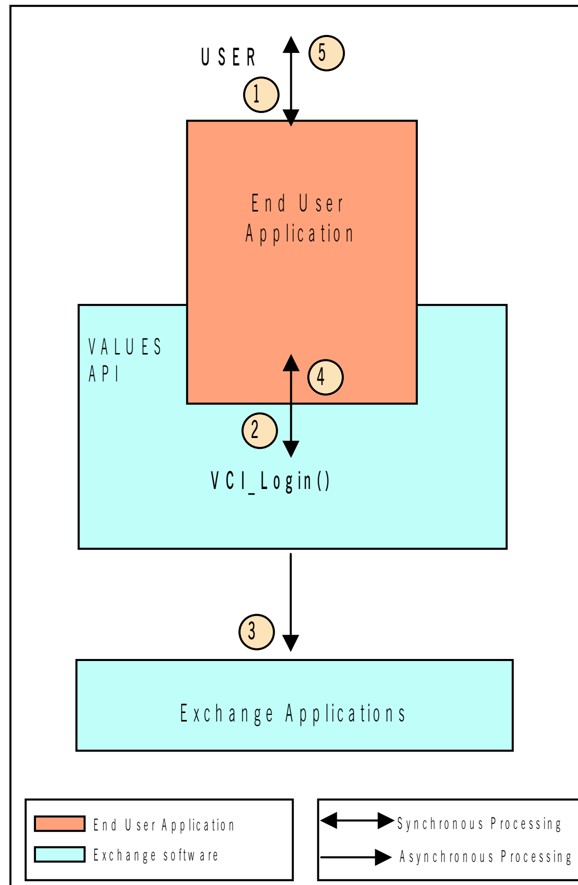


Figure 2.7 - Exchange Application Login

Step	Description	Input	Output
1	The end user application requests user ID and authorization data from the user.	n/a	UID, auth.
2	The application calls VCI_Login passing user ID, authorization data (password), dbApplID (target Xervice identifier) and application callback. The end user application waits for VCI_Login to return control.	UID, auth. data, dbApplID, application callback	n/a
3	VCI_Login forwards the login request to the Exchange application for processing. VCI_Login records the application callback for later use.	UID, auth. data,	n/a
4	VCI_Login returns to the calling application with the status of its transmission of the login request.	n/a	status
5	The end user application returns control to the user.	n/a	status

Table 2.5 - VCI_Login Process Flow

The login request has been transmitted to the Exchange application and an application callback has been scheduled. The exact processing status of the login request is unknown until a response is received.

2.3.3 Receiving a Login Response

In this section, the processing flow during receipt of a login response is described. The VCI_Dispatch entry point and application callback are explained in the context established by a login request (see *section 2.3.2*). *Figure 2.8* graphically depicts the processing involved in receiving a login response. The processing steps are described in *Table 2.6*

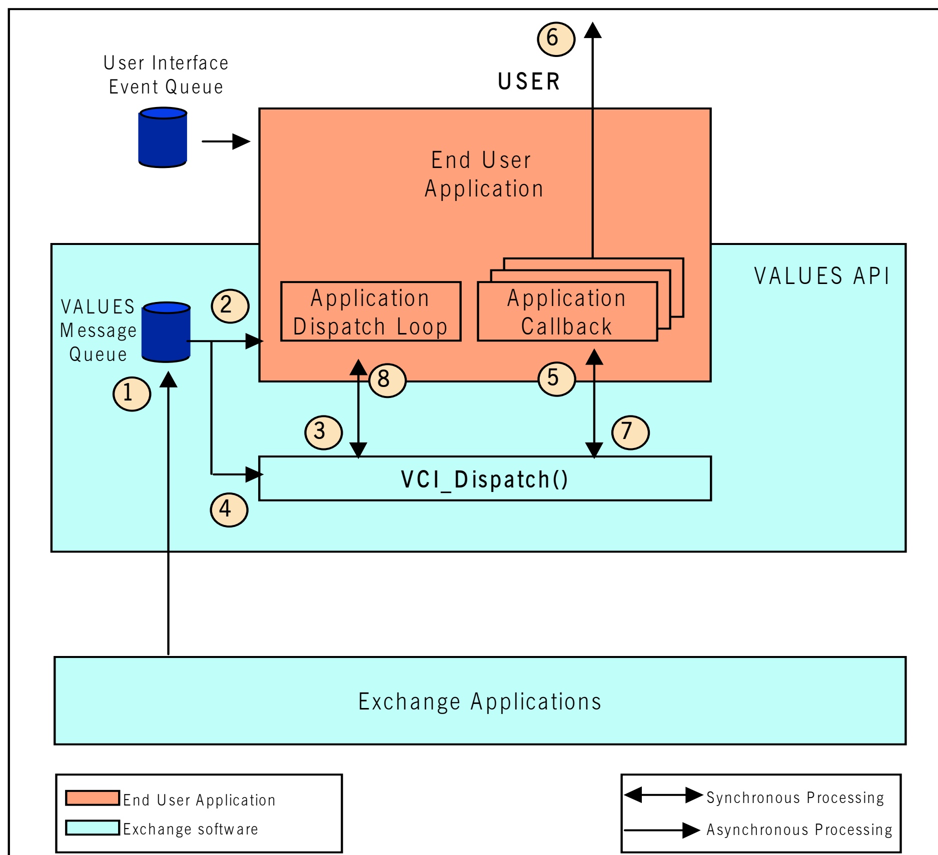


Figure 2.8 - Login Response Processing

Step	Description	Input	Output
1	A login response is sent to the VALUES message queue (VMQ) from an Exchange application.	response data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	response data	n/a
5	VCI_Dispatch identifies the application callback associated with the response message. VCI_Dispatch calls the application callback (i.e., the callback registered with the login request) passing a login ID and waits for the application callback to return.	status, login ID	status, login ID
6	The application callback recognizes that the message contains a login response by reading the status passed and displays the login status to the user. The application callback then stores the login ID for further reference.	status, login ID	status
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch deletes its knowledge of the login request context. VCI_Dispatch returns control to the application dispatch loop along with the status of its processing.	n/a	status

Table 2.6 - VCI_Dispatch Process Flow (Receipt of Login Response)

The end user application is now able to provide Exchange application functionality; e.g., order entry.

2.3.4 Logging Out from an Exchange Application Normally

In this section, the processing flow of a logout request is described. The VCI_Logout entry point is described in the context of a user terminating the end user application. It is assumed that during the session the user logged on to an Exchange application. The end user application can retain authorized communication to Exchange application throughout a session and perform all necessary logouts at session shutdown. Alternatively, the logout service can be used at any time during a session.

Figure 2.9 graphically depicts the processing involved in logging out from an Exchange application. The processing steps are explained in *Table 2.7*.

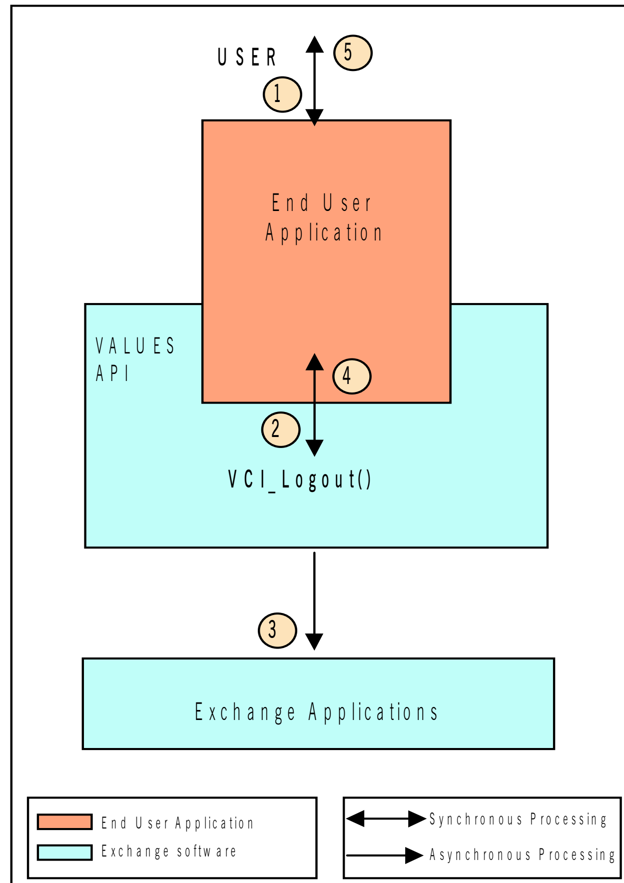


Figure 2.9 - Exchange Application Logout

Step	Description	Input	Output
1	The user requests a logout from the Exchange application.	n/a	n/a
2	The application calls VCI_Logout passing the dbApplID (Exchange application) and the login ID. The application waits for VCI_Logout to complete.	dbApplID, login ID	n/a
3	VCI_Logout forwards the logout request to the Exchange application.	user ID	n/a
4	VCI_Logout returns to the calling application with the status of its transmission of the logout request.	n/a	status
5	The application returns control to the user.	n/a	n/a

Table 2.7 - VCI_Logout Process Flow

The logout request has been transmitted to the Exchange application. The exact processing status of the logout request is unknown until a response is received.

2.3.5 Receiving a Logout Response

In this section the processing flow during receipt of a logout response is described. The VCI_Dispatch entry point and application callback are explained in the context established by a logout request (see also *section 2.3.4*). *Figure 2.10* graphically depicts the processing involved in receiving a logout response. The processing steps are described in *Table 2.8*.

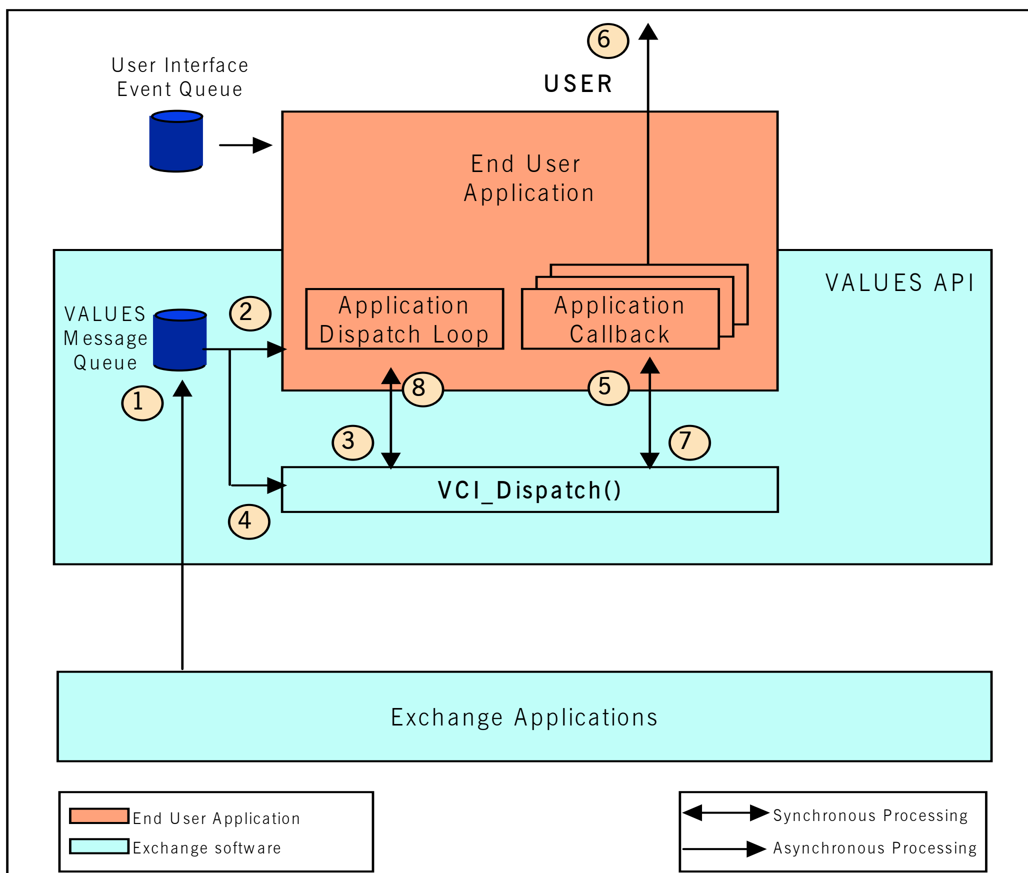


Figure 2.10 - Logout Response Processing

Step	Description	Input	Output
1	A logout response is sent to the VALUES message queue (VMQ) from the Exchange application.	response data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	response data	n/a
5	VCI_Dispatch identifies the application callback associated with the response message. VCI_Dispatch calls the application callback (i.e., for receipt of logout responses, the callback registered with the corresponding login request is used) passing a login ID and waits for the application callback to return.	status, login ID	status, login ID
6	The application callback recognizes that the message contains a logout response by reading the status passed and displays the logout status to the user. If applicable, it cleans up data related to the Exchange Application just logged off from.	status	status
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch deletes its knowledge of the logout request message, response message and application callback. VCI_Dispatch returns control to the application dispatch loop along with the status of its processing.	n/a	status

Table 2.8 - VCI_Dispatch Process Flow (Receipt of Logout Response)

The end user application has now successfully logged out from the Exchange application.

2.3.6 Logging Out from an Exchange Application Abnormally

A range of exceptions can occur when using GATE; i.e., failure of hardware and/or software components. If the application is unable to logout normally, exception handling must take place.

Exception handling is supported by VALUES API through completion status and application callback invocation. When an exception occurs, VALUES API responds with invocation of registered application callbacks in a defined sequence. Please refer to *section 3.2.3* for detailed information on exception handling.

2.4 Request Management Services

In this section, an overview of what requests are and how they are processed by the VALUES API is given. The entry point VCI_Submit and how it is used to send requests to Exchange applications is explained.

2.4.1 Overview

Request management services provide access to Exchange applications. A request consists of a single application request and a single application response.

An application request is used to trigger processing by an Exchange application. It consists of a request ID, which is unique per supported Exchange application, and any data needed to satisfy the request (e.g., order details for entering an order). For detailed information on specific application requests, please refer to the Exchange application specific volumes.

An application response contains the results of the Exchange application's processing. It is generated by the requested Exchange application and is delivered asynchronously to the end user application. An application response consists of the corresponding request ID, processing completion status, and any additional data needed to communicate processing results. Application responses will be received as a single response to a specific request.

2.4.2 Submitting an Application Request

In this section, the processing flow during submission of an application request is described. The VCI_Submit entry point is explained in the context of a user submitting an order. It is assumed that the user previously obtained authorization (for details on security and login see *section 2.3*).

Figure 2.11 graphically depicts the processing involved in submitting an application request. The processing steps are described in *Table 2.9*, using the application request "Enter Single Leg Order" as an example.

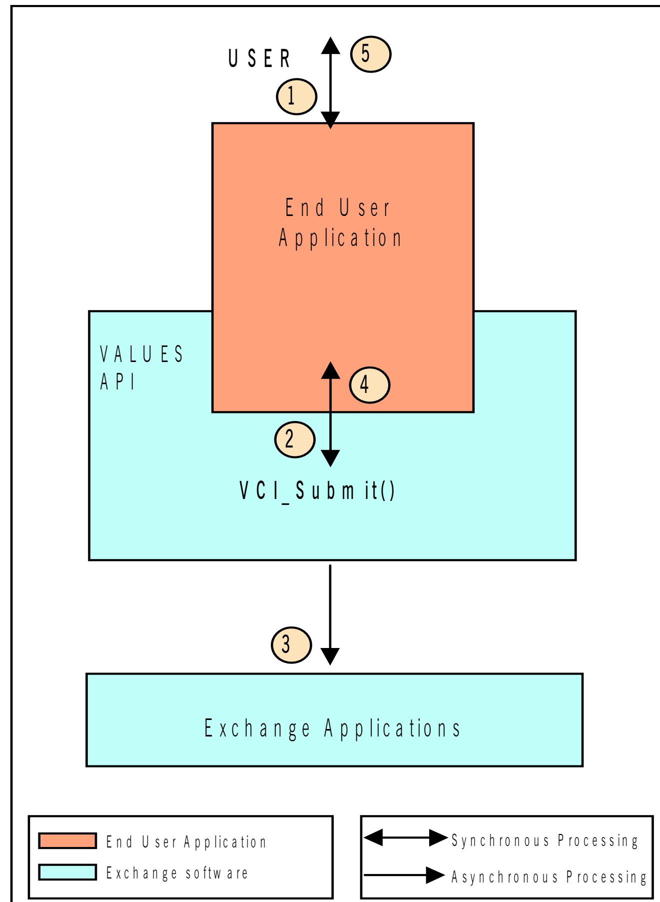


Figure 2.11 - Application Request Processing

Step	Description	Input	Output
1	User enters an order into his application.	order data	n/a
2	The end user application formats the "Enter Single Leg Order" application request and calls VCI_Submit to send it to the Exchange application. The end user application includes an application callback reference to be invoked on completion of the application request. The end user application waits for VCI_Submit to return control.	order data, callback reference, login ID, dbApplID	n/a
3	VCI_Submit forwards the application request to the Exchange application. VCI_Submit records the application callback for later use.	order data, dbApplID, user Id	n/a
4	VCI_Submit returns to the calling application with the status of its transmission of the application request.	n/a	status
5	The application returns control to the user.	n/a	n/a

Table 2.9 - VCI_Submit Process Flow

The user's order has been transmitted to the Exchange application and an end user application response callback has been scheduled. The exact processing status of the application request is unknown until a response is received.

2.4.3 Receiving an Application Response

In this section, the processing flow during receipt of an application response is described. The VCI_Dispatch entry point and application callback are explained in the context established by submission of an order (see *section 2.4.2*). *Figure 2.12* graphically depicts the processing involved in receiving an application response. The processing steps are described in *Table 2.10*.

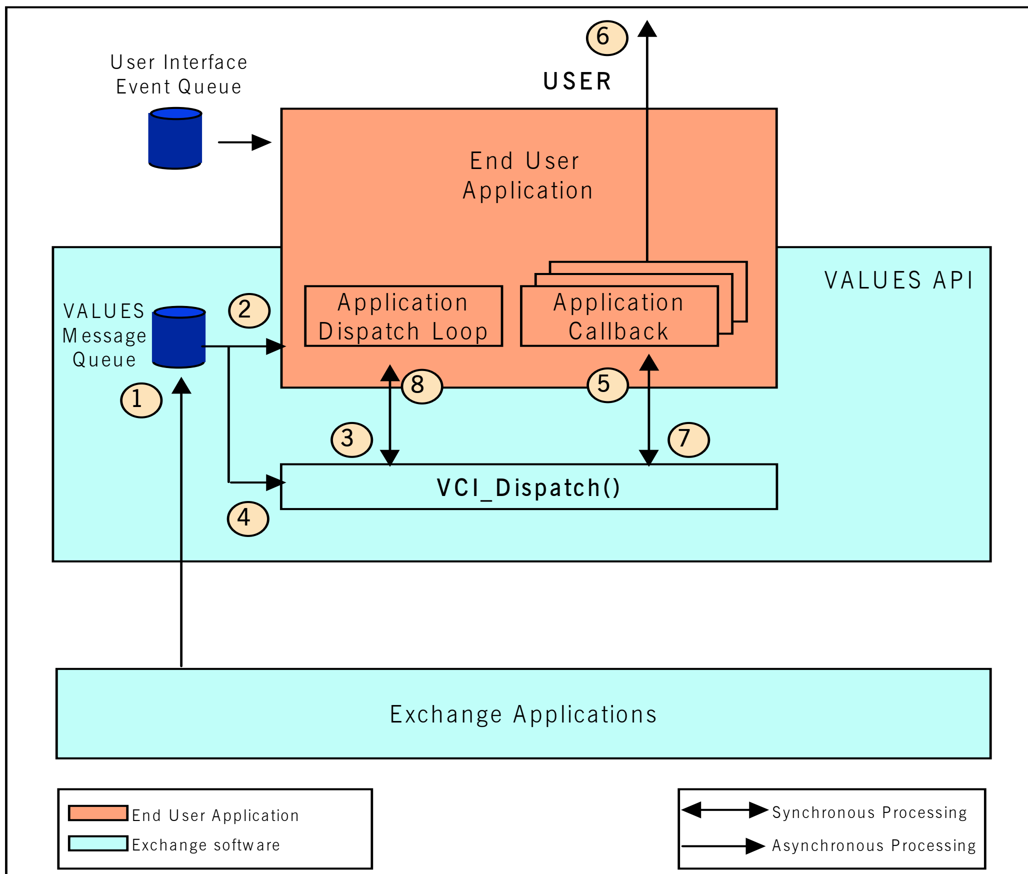


Figure 2.12 - Application Response Processing

Step	Description	Input	Output
1	An application response is sent to the VALUES message queue (VMQ) from an Exchange application.	response data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	response data	n/a
5	VCI_Dispatch identifies the application callback associated with the response message. VCI_Dispatch calls the application callback passing the application response and request data. VCI_Dispatch waits for the application callback to return.	response data, request data	n/a
6	The application callback displays the received data to the user.	n/a	n/a
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch deletes its knowledge of the application request message, response message and application callback. VCI_Dispatch returns control to the application dispatch loop along with the status of its processing.	n/a	status

Table 2.10 - VCI_Dispatch Process Flow (Receipt of Application Response)

The application has received the asynchronous response and displayed it to the user.

2.5 Subscription Management Services

This section starts with an overview of the subscription mechanism, data streams and broadcasting. The subsequent sections describe the Call Interface entry points to start a subscription, to receive subscription data and to terminate a subscription.

2.5.1 Overview

Subscription is the mechanism used to request event-driven information from the Exchange applications. Subscription can be started and stopped. Information requested through a subscription arrives asynchronously.

Note: As broadcast volumes can be very high, writing fast subscription callbacks is necessary. Please refer to *section 3.9.1* for details.

Subscriptions are supported on data streams (collections of selected data) which disseminate public and private data from the Exchange applications. Each Exchange will make its public data streams available to all exchange members in their market. Public market data such as “best prices” and “trade volume” or “news” are broadcast through public data streams. Private data streams are constructed individually for Exchange participants and contain information such as “own orders”.

Subscription to a data stream is valid for a certain period of time during a VALUES session. Subscriptions can be issued once or several times.

As soon as the user has subscribed to a data stream, broadcast data will be received continuously as it is generated by Exchange applications.

The services provided by subscription management are summarized in *Figure 2.13*.

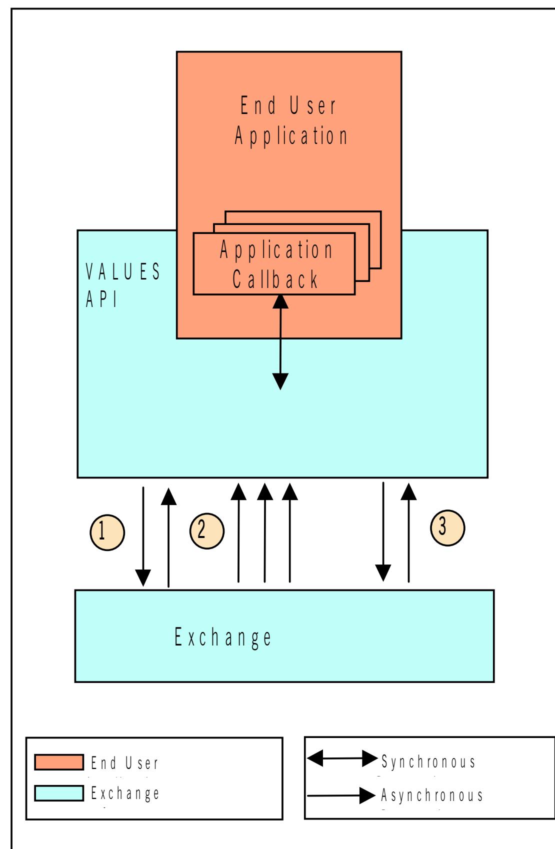


Figure 2.13 - Subscription Concept

Subscription requests are sent to the Exchange applications specifying the desired data stream, the corresponding subscription response is received asynchronously (1). Subsequently all new broadcast data on the specified stream is sent to the end user application asynchronously (2) via the callback specified with the subscription request. The end user application may stop the subscription at any time by sending an unsubscription request which is then confirmed by an asynchronous unsubscription response (3).

2.5.2 Broadcast Extension

Beginning with GATE Release 3.0, Exchange applications can implement the Broadcast Extension features. Please consult the Xervice-specific Volumes of VALUES API documentation for information if a given version of the Xervice implements Broadcast Extension.

In case Broadcast Extension is implemented, the following features are effective:

- Broadcast Authorization: For any broadcast subscription, a valid login context has to be established, much like in the Request Management concept. This feature is disabled for applications that use backward compatibility to the Call Interface Version using CVN_011.
- Broadcast Clustering: For selected high-volume broadcast streams, multiple messages of broadcast data may be clustered in a single VMQ message, thereby saving large quantities of I/O overhead. This feature is effective no matter which CVN is used by end user applications. Thus, it cannot be disabled by using GATE's backward compatibility mechanism.

2.5.3 Identifying Available Data Streams

Subscription to a data stream is made by specifying the subscription subject (Exchange Service, stream type, filter criteria) and a callback to receive subscription data when calling VCI_Subscribe. For detailed information on subjects and available data streams, please refer to the Exchange application specific volumes. The VALUES API Call Interface offers a generic recovery assistance service for broadcasts. For this purpose, a special subscription subject is available that enables the transmission of Gap Notifications within each stream it has been used on. For more details, please refer to *section 3.9*.

2.5.4 Subscribing to a Data Stream

In this section, the processing flow to subscribe to a data stream is described. The VCI_Subscribe entry point is explained in the context of a user opening a window. It is assumed that the user previously obtained authorization in case of a Xervice that implements Broadcast Extension (see *section 2.5.2*; for details on security and login see *section 2.3*).

The application has to pass a Xervice identifier(dbApplID) in order to specify the desired Exchange Service (=Xervice) when subscribing to a data stream. Please refer to *section 2.9* for information on how to obtain the correct value for dbApplID.

Figure 2.14 graphically depicts the processing involved in subscribing to a data stream. The processing steps are explained in *Table 2.11*

.

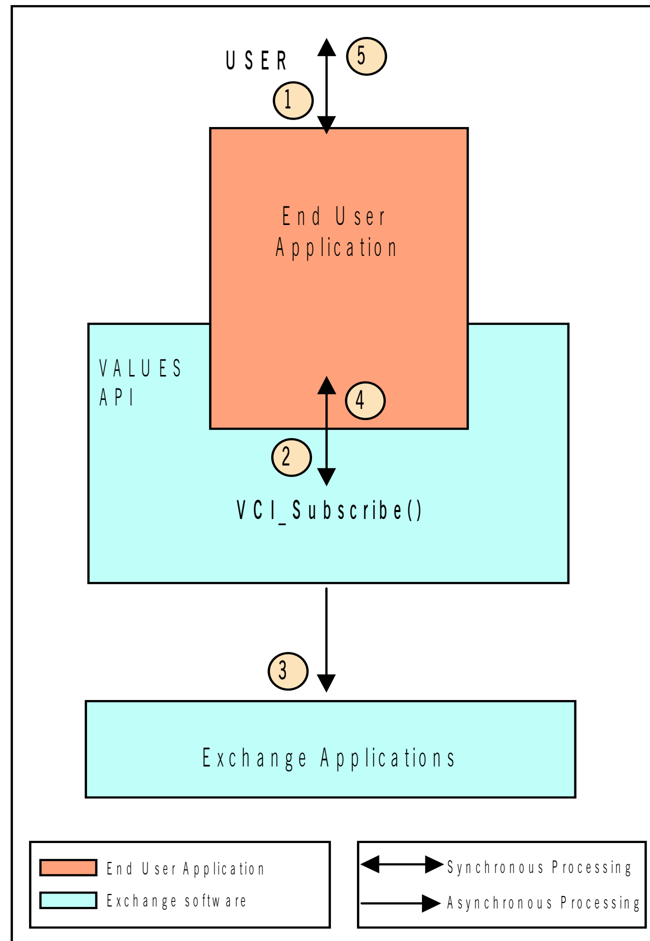


Figure 2.14 - Data Stream Subscription

Step	Description	Input	Output
1	A user opens a window to display market data (the application fetches an initial data set to fill the window, using an Inquiry)	n/a	n/a
2	The application calls the VCI_Subscribe entry point passing the data stream subject and an application callback reference. The end user application waits for VCI_Subscribe to complete.	subject, callback, dbApplID	n/a
3	VCI_Subscribe forwards the subscription request to the local architecture. VCI_Subscribe stores the application callback for later use.	subject, dbApplID	n/a

Table 2.11 - VCI_Subscribe Process Flow

Step	Description	Input	Output
4	VCI_Subscribe returns to the calling application with the status of its transmission of the subscription request.	n/a	status
5	The application returns control to the user.	n/a	n/a

Table 2.11 - VCI_Subscribe Process Flow

The subscription request has been transmitted to the Exchange application. The exact processing status of the subscription request is unknown until a subscription response is received.

The callback registered by VCI_Subscribe is used to handle

- a (single) Subscription Response (*section 2.5.5*)
- Subscription Data broadcasts (*section 2.5.6*).

2.5.5 Receiving Subscription Responses

In this section, the processing flow taking place when subscription responses arrive asynchronously is explained. Subscription responses indicate the status of a previously issued subscription request. The VCI_Dispatch entry point and application callback are described in the context established by a subscription request. *Figure 2.15* graphically depicts the processing involved in receiving subscription responses. The processing steps are explained in *Table 2.12*.

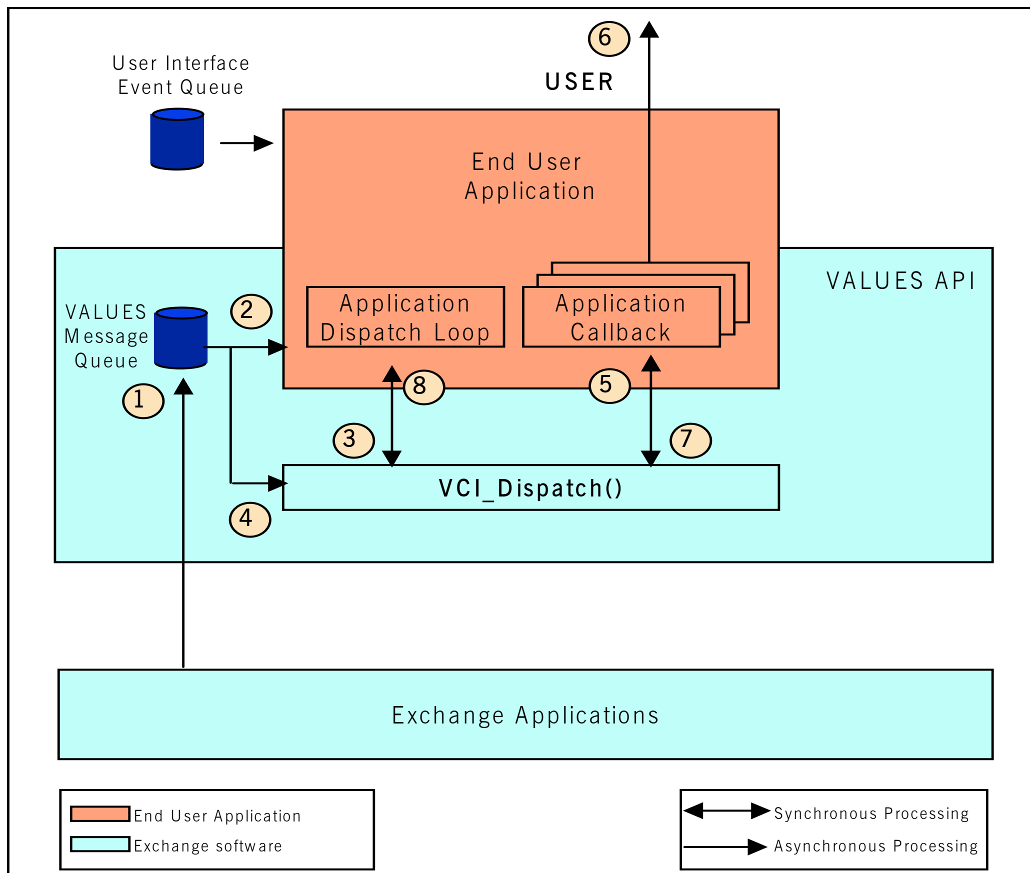


Figure 2.15 - Subscription Response Processing

Step	Description	Input	Output
1	A subscription response is sent to the VALUES message queue (VMQ) from the Exchange application.	response data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	response data [1]	n/a
5	VCI_Dispatch identifies the application callback associated with the subscription. VCI_Dispatch calls the application callback passing the subscription ID and status and waits for the callback to return.	status, subsID	status, subsID
6	The application callback recognizes that the message contains a subscription response by reading the status passed and displays the subscription status to the user. The application callback then stores the subscription ID for further reference.	status, subsID	status
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch passes control back to the application dispatch loop, returning the status of its processing.	n/a	status

Table 2.12 - VCI_Dispatch Process Flow (Receipt of Subscription Response)

1.Response data contains only the status of the subscription request, no broadcast data.

The end user application has subscribed to a data stream and will receive subscription data via the same registered application callback.

2.5.6 Receiving Subscription Data

In this section, the processing flow taking place when subscription data (i.e., broadcasts) arrives asynchronously is explained. The VCI_Dispatch entry point and application callback are described in the context established by a subscription. *Figure 2.16* graphically depicts the processing involved in receiving subscription data. The processing steps are explained in *Table 2.13*.

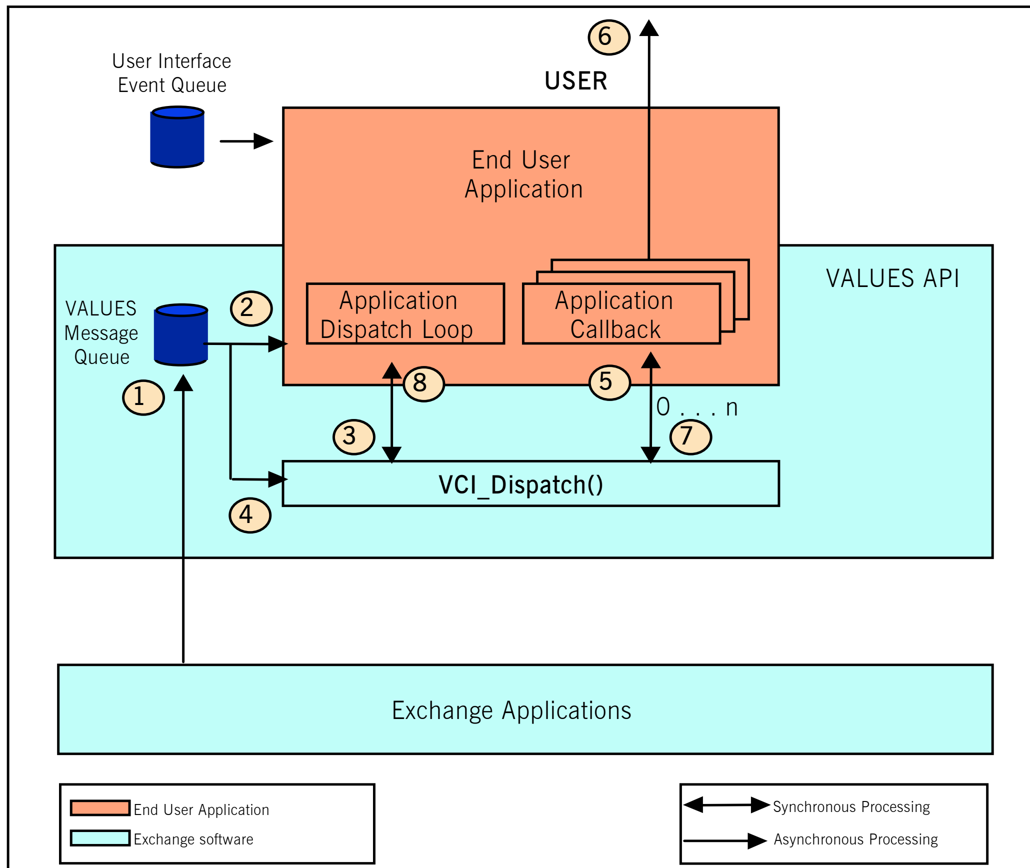


Figure 2.16 - Subscription Data Receipt

Step	Description	Input	Output
1	Subscription data is sent to the VALUES message queue (VMQ) from the Exchange application.	subscription data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	subscription data	n/a
5	VCI_Dispatch identifies the application callback associated with the subscription. VCI_Dispatch calls the application callback passing the subscription data and waits for the callback to return. Multiple callback invocations are possible. Under certain circumstances, it is possible that no callback invocation occurs at all, which is equivalent with VALUES discarding a VMQ message that is not required.	subscription data	subscription data
6	The application callback recognizes that the message contains subscription data by reading the status passed. The application callback displays the received data to the user, populating or updating the window which initiated the subscription.	n/a	subscription data
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch passes control back to the application dispatch loop, returning the status of its processing.	n/a	status

Table 2.13 - VCI_Dispatch Process Flow (Receipt of Subscription Data)

As long as the subscription is active, the callback will be invoked as new data arrives. The user may unsubscribe at any time or subscribe to other data streams.

2.5.7 Unsubscribing from a Data Stream Normally

In this section, the processing flow to stop a subscription is explained. The VCI_Unsubscribe entry point is explained in the context of a user closing a window which results in a request to stop the subscription. *Figure 2.17* graphically depicts the processing involved in unsubscribing. The processing steps are described in *Table 2.14*.

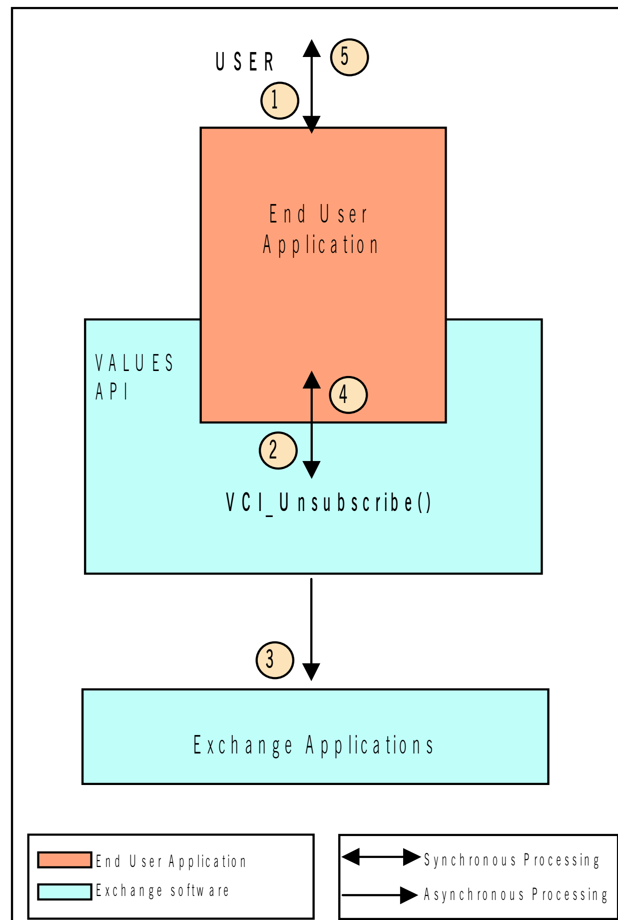


Figure 2.17 - Data Stream Unsubscription

Step	Description	Input	Output
1	The user closes the market overview window.	n/a	n/a
2	The application calls the VCI_Unsubscribe entry point passing the subscription ID. The end user application waits for VCI_Unsubscribe to complete.	subsID	n/a
3	VCI_Unsubscribe forwards the unsubscription request to the local architecture.	subsID	n/a
4	VCI_Unsubscribe returns to the calling application with the status of its transmission of the unsubscription request.	n/a	status
5	The application returns control to the user.	n/a	n/a

Table 2.14 - VCI_Unsubscribe Process Flow

The unsubscription request has been transmitted to the Exchange application. The exact processing status of the unsubscription request is unknown until a response is received.

2.5.8 Receiving Unsubscription Responses

This section explains the processing flow taking place when an unsubscription response arrives asynchronously. Unsubscription responses indicate the status of a previously issued unsubscription request. The VCI_Dispatch entry point and application callback are described in the context established by an unsubscription request. *Figure 2.18* graphically depicts the processing involved in receiving an unsubscription response. The processing steps are explained in *Table 2.15*.

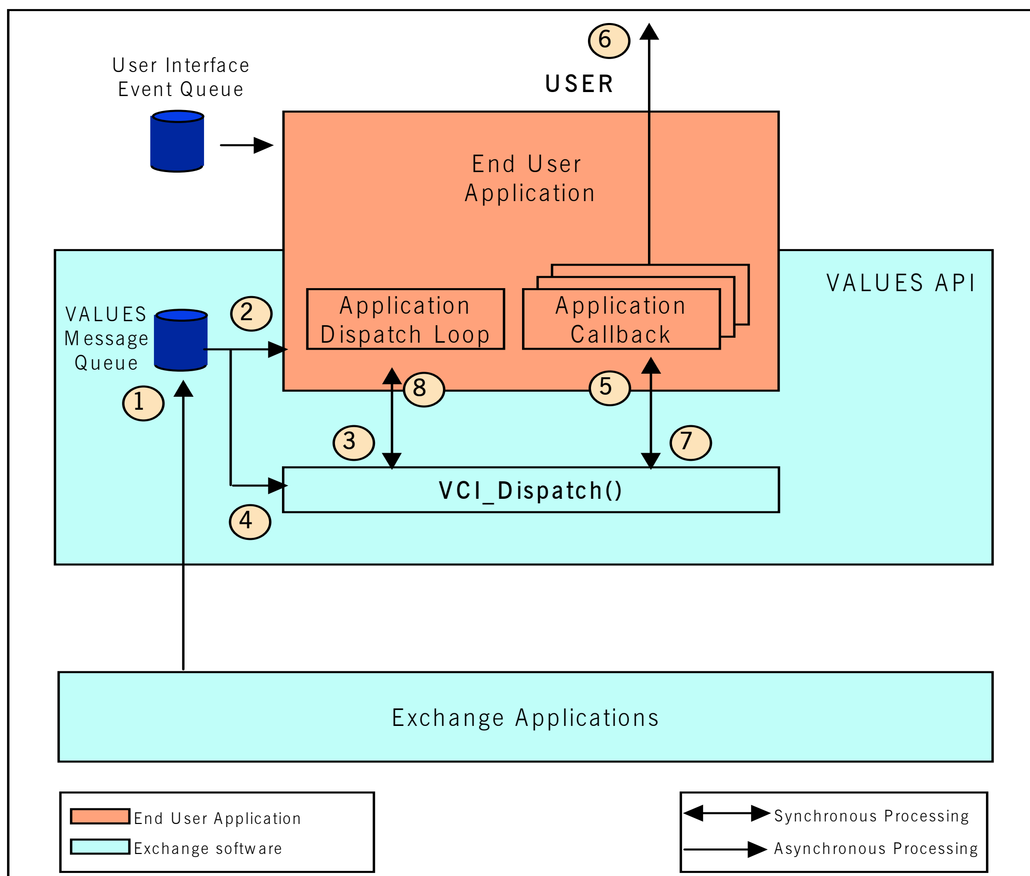


Figure 2.18 - Unsubscription Response Processing

Step	Description	Input	Output
1	An unsubscription response is sent to the VALUES message queue (VMQ) from the Exchange application.	response data	n/a
2	The application dispatch loop receives notification of an event having occurred on the VMQ.	VMQ event	n/a
3	The end user application calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the response message from the VMQ.	response data	n/a
5	VCI_Dispatch identifies the application callback associated with the corresponding subscription. VCI_Dispatch calls the application callback passing the subscription ID and status and waits for the callback to return.	status, subsID	status, subsID
6	The application callback recognizes that the message contains an unsubscription response by reading the status passed and displays the unsubscription status to the user. If applicable, it cleans up data related to the broadcast stream it just unsubscribed from.	status	status
7	The application callback returns to VCI_Dispatch.	n/a	n/a
8	VCI_Dispatch passes control back to the application dispatch loop, returning the status of its processing.	n/a	status

Table 2.15 - VCI_Dispatch Process Flow (Receipt of Unsubscription Responses)

The end user application has terminated subscription to a data stream.

2.5.9 Unsubscribing from a Data Stream Abnormally

A range of exceptions can occur when using GATE; i.e., failure of hardware and/or software components. If the application is unable to unsubscribe normally, exception handling must take place.

Exception handling is supported by VALUES API through completion status and application callback invocation. When an exception occurs, VALUES API responds with invocation of registered application callbacks in a defined sequence. Please refer to *section 3.2.3* for detailed information on exception handling.

2.6 Integrating VALUES Events

In this section, the mechanism for integrating VALUES events with the end user application is described. Integration is based on the receipt of VALUES events by the end user application and the forwarding of these events to VALUES API. The end user application is responsible for receiving VALUES events from a queue created by VCI_Connect on VALUES startup. The queue is provided by the operating system and based on the socket interface. The queue allows shared access for both VALUES and the end user application. The name of the message queue (socket identifier, VMQname) is returned by the VCI_Connect call to allow the end user application to access it. The application is expected to monitor the queue for events and dispatch them to the API as they occur.

There are four kinds of VALUES events:

- Response events are answers to end user application requests sent to Exchange applications via the VCI_Submit, VCI_Login, VCI_Logout, VCI_Subscribe and VCI_Unsubscribe entry point.
- Broadcast events are asynchronous data received after subscription to a data stream using the VCI_Subscribe entry point.
- Exception events are sent to VALUES in case of problems with the connection (e.g., network problems, GATE failure, etc.).
- Notification events are sent to VALUES in case of Exchange application availability state changes (e.g., initial Exchange application availability, when an Exchange application was restarted).

In order to avoid interruption of synchronous VALUES API calls (i.e. VCI_Connect, VCI_Disconnect) VALUES API-based application should disable asynchronous events (e.g. UNIX signals).

End user applications based on the VALUES API are generally in control of the process; i.e., the main processing loop is implemented in the end user application. The VALUES API is a library and does not contain a processing loop. Monitoring of events (e.g., VALUES events, keyboard input, mouse input) is therefore in responsibility of the end user application. The end user application dispatches (i.e., using VCI_Dispatch) process control to the VALUES API temporarily to allow handling of VALUES events.

Application callbacks are functions with a specific function prototype and a custom implementation. The function prototype of these callbacks are defined in the VALUES API specification, the end user application implements the callbacks. The end user application can register callbacks with most of the entry points by passing a callback reference. When the end user application receives a VALUES event it dispatches processing to the VALUES API. The VALUES API reads the data associated with the event from the VMQ, identifies the registered callback reference, formats the data associated with the event and invokes the callback passing the data to the end user application.

Figure 2.19 graphically depicts the processing involved in delivery of asynchronous responses and broadcast events as well as exceptions. The processing steps are described in *Table 2.16*.

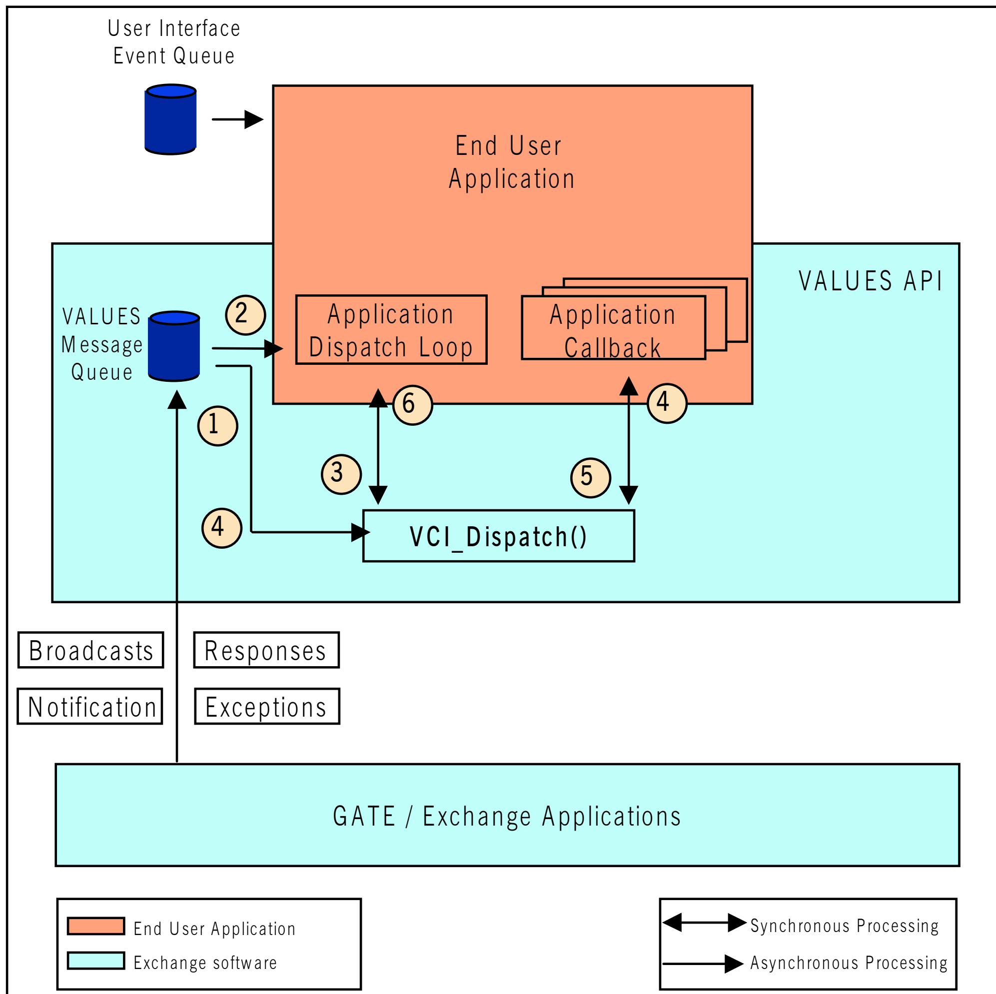


Figure 2.19 - VALUES Message Delivery to End User Application

Step	Description	Input	Output
1	Initiated by a VCI_Connect call, VALUES creates a queue (VMQ) which can be written to by another process. Exchange applications using GATE write VALUES events to the VMQ.	VALUES event	n/a
2	The end user application is responsible for checking the VMQ for events (e.g., by listening to VMQ events with the applications own event dispatch loop).	notification of pending event	n/a
3	When an end user application recognizes that an event has occurred on the VMQ, it calls VCI_Dispatch and waits for the call to complete.	n/a	n/a
4	VCI_Dispatch reads the message associated with the event from the VMQ and identifies the appropriate application callback to invoke. It is possible that multiple callback invocations take place, to the same or to various callback functions. VCI_Dispatch: Passes request and response data (in case of response) Passes broadcast data (in case of broadcast) Returns a notification (initial Exchange application state and in case of an Exchange application state change) Returns an exception (in case an exception occurred) Invokes pending callbacks (in case of disconnect and logout, Exchange application unavailability and non-transparent failover)	VALUES message	n/a
5	The application callback determines the type of message received (i.e., responses, broadcasts, notofications or exceptions) by reading the status passed. The application callback then displays the received information and returns to VCI_Dispatch.	status	received information
6	VCI_Dispatch returns status to the application event dispatch loop. The application resumes processing.	n/a	status

Table 2.16 - VCI_Connect and VCI_Dispatch Process Flow (VALUES Message Delivery)

2.7 Recovery Management Services

Some Exchange applications support recovery of application requests, application responses and broadcasts. For a detailed description of these recovery management services please refer to the Exchange application specific volumes.

For broadcasts, the VALUES API Call Interface offers a generic recovery assistance service. A special subscription subject is available that enables the transmission of Gap Notifications within each stream it has been used on. Please refer to *section 3.9* for details on how to subscribe for Gap Notifications.

For recoverable broadcasts, usage of the Exchange application-specific recovery mechanisms should be the preferred method.

2.8 Multi-User Capability

One instance of the VALUES API supports simultaneous access for multiple users of multiple Exchange applications.

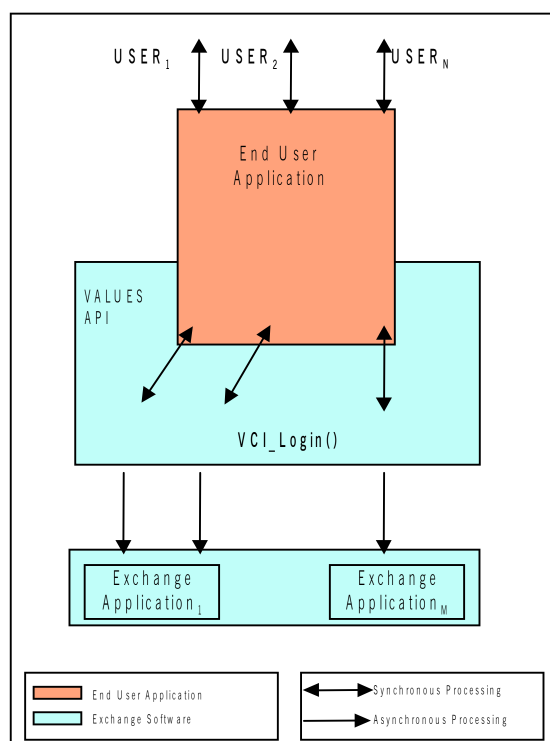


Figure 2.20 - Multi-User Login

At login (by receiving the login response), the end user application receives a login ID from the VALUES API. The login ID is unique for each successful call to VCI_Login and a specific Exchange application.

In subsequent calls to the entry points VCI_Logout, VCI_Subscribe (in case of Xervices with Broadcast Extension, compare *section 2.5.2*) and VCI_Submit, the end user application must specify this login ID to authenticate the specific user.

The callback registered with each call to VCI_Submit, VCI_Subscribe (Broadcast Extension only, see *section 2.5.2*) or VCI_Login returns the login ID to the end user application. The end user application can dispatch responses to the specific user based on the received login ID.

It is the responsibility of the application using the multi-user capability to ensure appropriate and authorized use of login IDs obtained from the VALUES API.

2.9 Xervices, Xervice Classes and Multi-Exchange Capability

A Xervice is a service offered to a VALUES-based Front End application by a certain electronic exchange. A single exchange can offer multiple Xervices, e.g. Xetra Frankfurt offers a Xervice for transactions, a Xervice for retransmission of (recoverable) broadcasts and a Xervice used for continous auction quoting. To obtain access to a Xervice, an application has to log in to it, using VCI_Login. The Xervice identifier (dbApplID, see *section 6.2.20*) is used to specify the desired Xervice within this call.

Xervice identification data is available at run-time, as soon as a GATE session has been established. The data supplied by GATE includes Exchange identification information (Market Identification code (MIC), descriptive text). Thus an application can select an exchange or offer readable information to a user. This feature allows Front End applications to integrate new Exchanges that were not yet known at compile-time, using an unchanged executable. This holds true only if the new Exchange offers VALUES functionality that is identical to the existing Exchange for that the application was designed and built.

A Front End application can determine if Xervices offered by different electronic exchanges offer identical VALUES functionality by evaluating Xervice Class information. A Xervice Class is an identifier that is equal across Xervices which offer identical VALUES functionality, given the Application Version Number AVN is also equal. For example, if a Front End application was built to work with a certain exchange using two Xervices, their Xervice Classes being XETRA_TXN_XCLASS and XETRA_CAQ_XCLASS with AVN_xyz, it is able to work with any other exchange that offers the same Xervice Classes and AVN. The Xervice Class identifier is supplied in the field applClass of XerviceInfoT (see *section 6.2.2*).

GATE supplies Xervice Class information and Xervice identification data via the callback registered when using VCI_Connect. Please refer to *section 3.3.2* for details.

2.10 VALUES API Backwards Compatibility Concepts

In this section, an overview on backwards compatibility concept of VALUES API is given. Call Interface specific concepts are described in this Volume, while exchange specific concepts are explained in the related exchange specific volume.

The VALUES API supports backwards compatibility which is defined as support for the current version and the previous version of application requests, broadcast data structures and the Call Interface. The backwards compatibility of the Call Interface and the Exchange applications are independent of each other.

- Call Interface (technical components)
  The VALUES API supports the Call Interface of the current VALUES API Call Interface release (Call Interface Version Number CVN $_{x}$) and is backwards compatible to the Call Interface of the previous VALUES API Call Interface release (CVN $_{x-1}$). Each Call Interface entry point must contain a CVN which defines the version of the Call Interface to be used. All Call Interface entry points submitted within a VALUES session must correspond to the version specified in the connection request.
- Application requests and subscription requests (functional components)
  The VALUES API can support application requests of the current release (Application Version Number AVN $_{x}$) and be backwards compatible to the previous release (AVN $_{x-1}$) at the same time. Please refer to the Exchange application specific volumes to determine if the desired Exchange application release offers backwards compatibility. Each login and subscription request must contain an exchange-specific application version number (AVN) which defines the version of the associated

Exchange application to be used. All application requests submitted within a login must correspond to the version specified in the login request.

The version number CVN is published with each VALUES API Call Interface release. The AVN will be published with each release of the Exchange application. The end user application must use these numbers to specify the version it intends to use; i.e., the CVN has to be sent with each call of an entry point and the AVN with each login and subscription request.

Backwards compatibility cannot be guaranteed under all circumstances due to the potential for changes driven by legal imperatives, technology evolution etc. This means:

- Most updates of the VALUES API will not require re-compilation and re-linking of end user applications.
- Some updates of the VALUES API may require re-compilation and re-linking of end user applications.
- In future releases of Exchange applications or GATE, code changes, re-compilation and re-linking of end user applications may be required.

3 VALUES API Call Interface Reference

3.1 Overview

This section describes the structure of each entry point to the interface, including entry point name, description and calling syntax. Also, a list of all parameters of which the entry points are comprised is given. Parameters can be structures in which case a list of fields is given.

Each field listed is marked to show whether it is mandatory (m), optional (o), read-only (ro), or reserved (r) in the column headed "Usage". Also, fields are marked to show whether they are to be filled by VALUES (V) or by the end user application (A) in the column headed "Filled By". Fields to be filled by the end user application may be changed by VALUES. The character "-" in either column indicates that the information is not applicable for the given field.

Detailed field descriptions are given in *section 6*. The field descriptions contain information on where the fields are used, field characteristics such as data type and passing mechanism. The field description also defines the rules that apply for the individual data fields.

This section describes the following entry points:

- VCI_Connect
- VCI_Disconnect
- VCI_Dispatch
- VCI_Login
- VCI_Logout
- VCI_Submit
- VCI_Subscribe
- VCI_Unsubscribe.

Subsequently the interface specification for application callbacks is described. VALUES API provides a function type which must be used to declare the application callbacks referenced in VCI_Connect, VCI_Login, VCI_Submit and VCI_Subscribe.

All interface parameters detailed in the manual are pointers to data structures. The parameter *callbackCntxtData* (application callback context data) and those parameters prefixed “req”(request) are pointers to data structures populated by the end user application and passed to the Call Interface entry point.

The first parameter of each entry point is *reqControl*. The *reqControl* structure combines data fields that are common to all entry points and used as Call Interface control information.

Note: It is strongly advised to always use the *ReqCntrlT* data type to access or copy the fields of reqControl.

The parameter *statusData* and those parameters prefixed “resp” (response) denote pointers to data structures populated by the Call Interface entry point and returned to the end user application. Exceptions are the application callbacks, refer to *section 3.11* for detailed description on population of the data structures of application callbacks.

The calling application is responsible for allocation (and de-allocation) of memory sufficient to hold the different data structures. Please refer to *section 4* for details. The VALUES API Call Interface stores only the pointers to the request and the context data. The application callback parameter must be populated with a reference to a function. This function’s signature must comply to the application callback interface specification (see *section 3.11*).

It is recommended to initialize all reserved or unused optional data fields passed to the entry points. A guideline is given in *section 6.1.2*.

Each entry point returns a status data structure to the VALUES API-based application, containing the completion status of a call. The following table lists and describes the fields of the status data structure:

Exception Source	Field	Description
Exchange application	complSeverity	Severity of an exception.
	complCode	Code which uniquely identifies an exception within the context of an Exchange application.
	complText	Decode, or textual representation of the exception.
GATE,	techComplSeverity	Severity of an exception.
VALUES API	techComplCode	Code which uniquely identifies an exception.
	techComplText	Decode, or textual representation of the exception.

Table 3.1 - The Call Interface Status Data Structure

Note: The Exchange application specific completion status is applicable only if the technical completion code (*techComplCode*) indicates an unsuccessful request processing.

Depending on the severity of an exception, the end user application should perform different levels of exception processing.

The following table describes the different levels of severity for exception codes.

Field	Value	Description
complSeverity,	VCI_SUCCESS	Successful completion.
techComplSeverity	VCI_WARNING	A minor error has occurred. The application may have to perform error-handling processing depending on which completion code has been returned.
	VCI_ERROR	An error has been detected. The application must perform error-handling processing depending on which completion code was returned.
	VCI_FATAL	A fatal error has occurred. The application should perform a shutdown.

Table 3.2 - Call Interface Exception Severity Classes

For each entry point, a list of the relevant completion codes and their corresponding message text is provided. For the complete list of technical completion codes see *section 5*. Functional completion codes are provided with the Exchange specific volumes.

3.2 State Diagrams

3.2.1 Overview

The VALUES API state diagrams describe which VALUES API calls can be invoked depending on the current state of the application's use of VALUES API.

State transitions can be either VALUES API calls invoked by the end user application or application callbacks invoked by VALUES.

The notation 1..s indicating multiple possible subscriptions, 1..u indicating multiple users and 1..x indicating multiple Xervices are used for the respective states.

The following sections provide state diagrams for each of the VALUES service categories. State diagrams for normal operation and description of exception cases are provided.

3.2.2 Normal Operation

Session Management Services

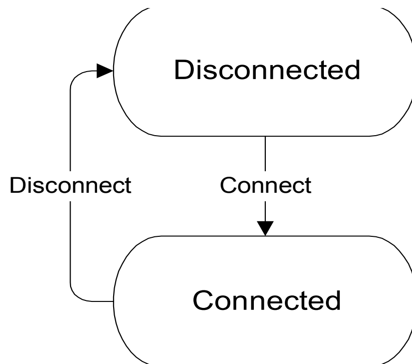
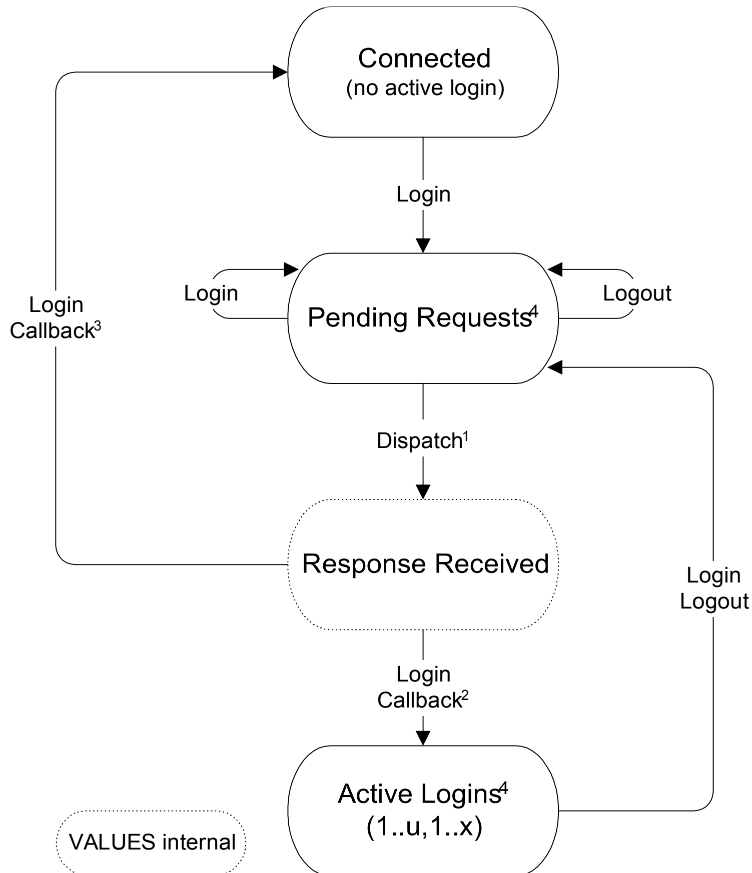


Figure 3.1 - State Diagram for Session Management Services

Security Management Services



[1] Dispatch is called by the application after notification via VMQ.
[2] Callback invoked by VALUES at receipt of a login or logout response.
[3] Callback invoked by VALUES at receipt of the last logout response (i.e., no more active logins).
[4] The states "Active Logins" and "Pending Requests" can be active simultaneously, in this case transitions are possible from both states.

Figure 3.2 - State Diagram for Security Management Services

Request Management Services

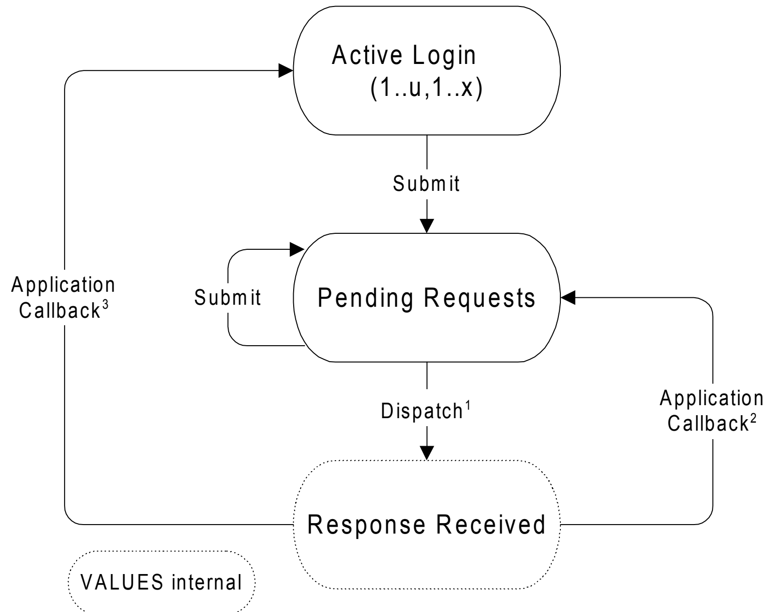
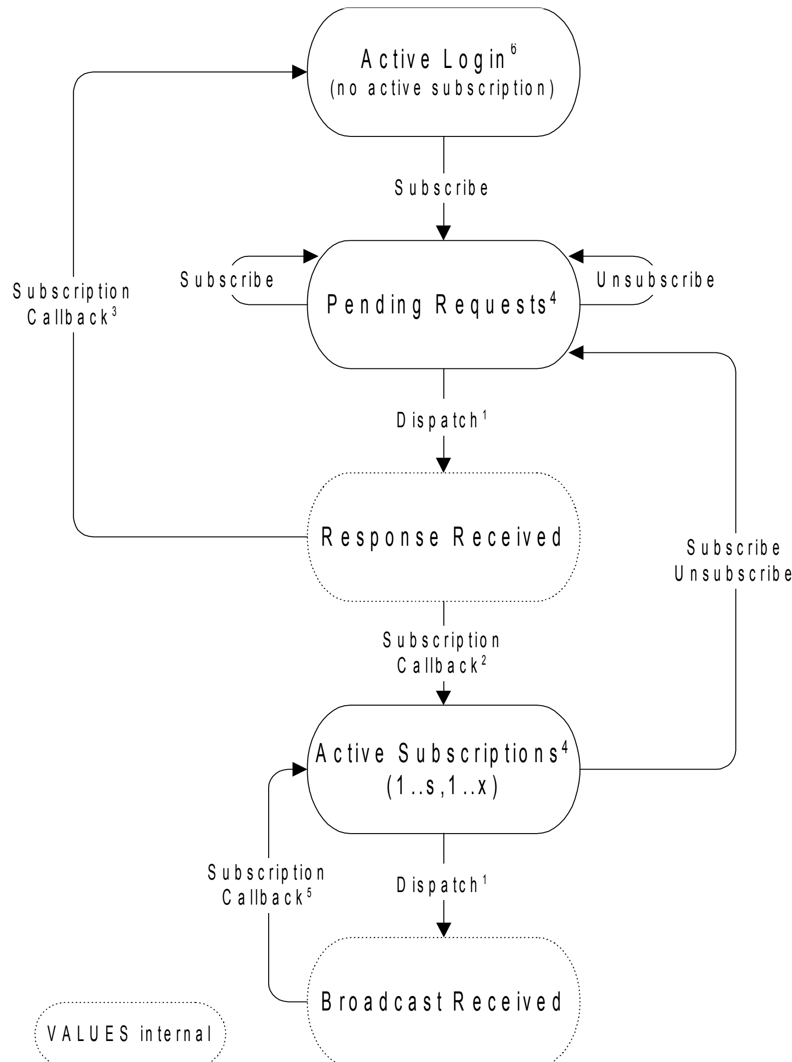


Figure 3.3 - State Diagram for Request Management Services

Subscription Management Services (Broadcast Extension assumed)



[1] Dispatch is called by the application after notification via VMQ.
[2] Callback invoked by VALUES at receipt of a subscription or unsubscription response.
[3] Callback invoked by VALUES at receipt of the last unsubscription response (i.e., no more active subscriptions).
[4] The states "Active Subscriptions" and "Pending Requests" can be active simultaneously, in this case transitions are possible from both states.
[5] Callback invoked by VALUES at receipt of a broadcast message.
[6] Compare the Broadcast Extension section

Figure 3.4 - State Diagram for Subscription Management Services

3.2.3 Exception Handling

A range of exceptions can occur when using the VALUES API. VALUES API exception handling performs a state change if applicable and notifies the end user application either by returning a status, or by invoking one or more callbacks.

In the following tables, the main exception cases are listed with the VALUES state after the exception is handled and a description of the notification mechanism (including the techComplSeverity and techComplCode) is provided.

Exception	Resulting State
The end user application disconnects from VALUES while requests are pending, logins are active, or subscriptions are active.	Disconnected

Notification Mechanism:

Callbacks are invoked in sequence:

1. Login callback – for each active login
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_XERVICE_NOT_AVAILABLE
2. Submit callback – for each pending request (including login, logout, subscribe and unsubscribe)
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_PENDING_REQUEST_DELETED
3. Subscribe callback – for each active subscription
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_SUBSCRIPTION_DELETED
4. Connection callback
   depending on disconnection response completion code either
   techComplSeverity: VCI_SUCCESS
   techComplCode: ELB_TECH_OK
   or
   techComplSeverity: VCI_ERROR
   techComplCode: ELB_TECH_REQ_UNSUCCESSFUL

Exception	**Resulting State**
The end user application performs a logout while requests are pending.	Connected

Notification Mechanism:

Callbacks are invoked in sequence:

1. Login callback – for the specific login
   techComplSeverity: VCI_SUCCESS
   techComplCode: ELB_TECH_LOGGED_OUT
2. Submit callback – for each pending request of the specific login
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_ PENDING_REQUEST_DELETED
3. Subscribe callback – for each pending subscription request of the specific login, if Broadcast Extension is implemented by the Xervice (see *section 2.5.2*).
4. techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_ PENDING_REQUEST_DELETED
5. Subscribe callback – for each active subscription of the specific login
6. techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_SUBSCRIPTION _DELETED
7. Connection callback
   techComplSeverity: VCI_SUCCESS
   techComplCode: ELB_TECH_LOGGED_OUT

Exception	**Resulting State**
Improper use of entry points (i.e., state transition not displayed in the state diagrams for normal operation) or invalid parameters passed to entry points.	No change

Notification Mechanism:

The entry point returns with a status indicating the exception.

Exception	Resulting State
Failed validation of application request data.	No change
Notification Mechanism: The submit callback is invoked with a status indicating the exception.	

Exception	Resulting State
Exchange Service unavailable.	Active Logins / Connected[1]
Notification Mechanism: Callbacks are invoked in sequence: 1. Login callback – for each active login of the specific Exchange Service techComplSeverity: VCI_WARNING techComplCode: ELB_TECH_XERVICE_NOT_AVAILABLE 2. Submit, Login, Subscribe(Broadcast Extension only) callback – per pending request of the specific Exchange Service (including submit, login, logout, subscribe and unsubscribe) techComplSeverity: VCI_WARNING techComplCode: ELB_TECH_ PENDING_REQUEST_DELETED 3. Subscription callback – per active subscription of the specific Exchange. 4. techComplSeverity: VCI_WARNING techComplCode: ELB_TECH_ SUBSCRIPTION_DELETED 5. Connection callback techComplSeverity: VCI_SUCCESS techComplCode: ELB_TECH_XERVICE_NOT_AVAILABLE The associated Exchange Service is specified in dbApplID of reqControl.	

1. If all Exchange applications are unavailable the resulting state is “Connected”.

Exception	Resulting State
GATE (Technical Services) unavailable	Disconnected

Notification Mechanism:

Callbacks are invoked in sequence:

1. Login callback – for each active login
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_XERVICE_NOT_AVAILABLE
2. Submit callback – for each pending request (including login, logout, subscribe and unsubscribe)
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_ PENDING_REQUEST_DELETED
3. Subscribe callback – for each active subscription
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_SUBSCRIPTION_DELETED
4. Connection callback
   techComplSeverity: VCI_FATAL
   techComplCode: ELB_TECH_TECHSRCV_NOT_AVAILABLE

Exception	Resulting State
MISS non-transparent failover for an Exchange Service which does not support retransmission of application requests.	Active Logins/ Connected

Notification Mechanism:

Callbacks are invoked in sequence:

1. Login callback – for each active login of the specific Exchange Service
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_NONTRANSPARENT_FAILOVER
2. Submit/Login/Subscribe callback – per pending request of the specific Exchange Service request (including submit, login, logout, subscribe and unsubscribe)
   techComplSeverity: VCI_WARNING
   techComplCode: ELB_TECH_ PENDING_REQUEST_DELETED
3. Connection callback
   techComplSeverity: VCI_SUCCESS
   techComplCode: ELB_TECH_NONTRANSPARENT_FAILOVER

The associated Exchange Service is specified in dbApplID of reqControl.

3.3 VCI_Connect

3.3.1 Overview

Description

This Call Interface entry point is used to connect to GATE. VCI_Connect initiates and performs linkage of the application with VALUES to send requests to Exchange Services and receive responses. VCI_Connect is the entry point which initiates a VALUES session. After a successful VCI_Connect, the application can send login requests to Exchange Services. When using a Xervice that does not implement Broadcast Extension (see *section 2.5.2*), it is also possible to subscribe to broadcast streams, without a previous login.

VCI_Connect requires registration of an application callback to receive notification and exceptions (e.g., Exchange Service availability). Associated with the callback, the application can specify custom context data which is buffered by VALUES and returned to the application when the callback is invoked. Only the address of the context data specified by End User application is stored by VALUES. Typically, VCI_Connect is invoked at startup of the application.

VCI_Connect is a synchronous request which, if successful, returns the name of the VALUES message queue (VMQname) which the application must monitor. In order to avoid interruption of VCI_Connect, asynchronous events (e.g. UNIX signals) should be disabled by the application.

Note: The VMQ must be monitored for events immediately after successful session establishment, and monitoring must be sustained until disconnecting the session.

For the Call Interface field descriptions please refer to *section 6*.

Syntax

```
void VCI_Connect (
        ReqCntrlT           *reqControl,
        CnctReqDataT        *reqData,
        AppCallbackT        *callbackFunc,
        AppCntxtDataT       *callbackCntxtData,
        CnctRespDataT       *respData,
        StatusDataT         *statusDataGlobal
        );
```

Parameter Name	**Fields**	**Usage**	**Filled By**
reqControl	VCIver (see *section 3.3.2*)	m	A
	connectionID	r	-
	dbApplID	r	-
	loginID	r	-
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
reqData	userID	m	A
	password	m	A
callbackFunc	(reference to the application exception callback function, see *section 3.3.3*)	m	A
callbackCntxtData	custBlockSize	m	A
	custData	o	A
respData	connectionID	ro	V
	VMQname	ro	V
	prodMode	ro	V
statusDataGlobal	complSeverity	ro	V
	complCode	ro	V
	complText	ro	V
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

Status Field Name	Value	Text
complSeverity,	VCI_SUCCESS	
techComplSeverity	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode,	ELB_TECH_INVALID_PASSWORD	INVALID PASSWORD
complText	ELB_TECH_INVALID_USER	USER IS NOT ALLOWED TO USE THE EXCHANGE SERVICE OR USER IS NOT REGISTERED
	ELB_TECH_INVALID_USER_OR_ PASSWORD	INVALID USER OR PASSWORD[1]
	ELB_TECH_BADLY_FORMED_ PARAMETER	A PARAMETER WAS EITHER INVALID OR HAD INVALID CHARACTERS[a]
techComplCode,	ELB_TECH_OK	SUCCESSFUL COMPLETION
techComplText	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_ALREADY_ CONNECTED	APPLICATION ALREADY CONNECTED
	ELB_TECH_CONNECTION_LIMIT_ REACHED	MAXIMUM NUMBER OF CONNEC-TIONS REACHED
	ELB_TECH_REQ_UNSUCCESSFUL	REQUEST NOT SUCCESSFULLY PROCESSED
	ELB_TECH_TECHSRV_NOT_ AVAILABLE	TECHNICAL SERVICES NOT AVAILABLE
	ELB_TECH_XERVICE_NOT_ AVAILABLE	EXCHANGE SERVICE NOT AVAILABLE
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

1.Applies to Windows platforms only.

3.3.2 VALUES Call Interface Version

The VALUES API conceptually supports backwards compatibility of the Call Interface. This document describes version 1.2 of the VALUES API Call Interface. The VALUES Call Interface Vresion VCIver can be set to CVN_xxx where "xxx" is depending on which version the end user application intends to use. The specific constant CVN_xxx is defined in the header Values.h which is delivered with the VALUES API header package.

With GATE Release 3.5, the Call Interface Version constant to be used is CVN_012. This is the same Call Interface Version as used since GATE Release 3.1.

Note: Within one VALUES session the same CVN must be used for all entry points. This is checked by VALUES.

3.3.3 The Connect Application Callback

The connect application callback is used to notify the end user application of exceptions (e.g., GATE unavailable). It is also used to notify the end user application of Exchange Service availability state changes (i.e., Xervice available or unavailable). After successful completion of VCI_Connect the connect application callback is invoked by VALUES for each Exchange Service available.

Please refer to *section 3.11* for details on the application callback function type, parameter, and completion code.

Notification of exceptions and availibility state changes are returned with the status data of the connect application callback. In case of an event associated to a certain Xervice (see *section 2.9* for a description of the Xervice concept), the following information is passed to the callback function (as a pointer to a struct of type XerviceInfoT):

Parameter Name	Fields	Usage	Filled By
appRespData	applClass	-	V
	applVersion	-	V
	applPrevVersion	-	V
	exchApplId	-	V
	exchDscrName	-	V

This applies to callbacks invoked with the following completion codes (techComplCod):

- ELB_TECH_XERVICE_AVAILABLE
- ELB_TECH_XERVICE_NOT_AVAILABLE
- ELB_TECH_LOGGED_OUT
- ELB_TECH_NONTRANSPARENT_FAILOVER

Please note that the dbApplID identifying the Xervice that has triggered the event is always available from the Request control structure (reqControl->dbApplID) in these cases.

3.4 VCI_Disconnect

Description

This Call Interface entry point is used by the end user application to disconnect from a VALUES session. VCI_Disconnect is typically called at shutdown of the end user application to terminate a session.

VCI_Disconnect is a synchronous request. In order to avoid interruption of VCI_Disconnect, asynchronous events (e.g. UNIX signals) should be disabled by the application.

Syntax

```
void VCI_Disconnect(
        ReqCntrlT          *reqControl,
        DiscnctReqDataT    *reqData,
        StatusDataT        *statusDataGlobal
        );
```

Parameter Name	**Fields**	**Usage**	**Filled By**
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	r	-
	loginID	r	-
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
reqData	n/a	r	-
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

Status Field Name	Value	Text
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode, complText	n/a	
techComplCode, techComplText	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.5 VCI_Dispatch

Description

This Call Interface entry point is used by the end user application to service Exchange Service responses, broadcast data, notifications, and exceptions. The end user application is responsible for checking the VMQ for events. In case of any event, the application should call VCI_Dispatch. VCI_Dispatch will read the associated data from the VALUES message queue and pass it on to the appropriate application callback. VCI_Dispatch is a synchronous request, which returns to the calling end user application on completion of the invocation of application callback(s).

VCI_Dispatch identifies the application callback to be invoked based on a VALUES-internal table in which each application connection, login, request, subscription and the associated callback are stored. For more information on how to integrate VALUES events please refer to *section 2.6*.

Syntax

```
void VCI_Dispatch(
        ReqCntrlT          *reqControl,
        StatusDataT        *statusDataGlobal
        );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	r	-
	loginID	r	-
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

Status Field Name	Value	Text
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode,	n/a	

complText

techComplCode,	ELB_TECH_OK	SUCCESSFUL COMPLETION
techComplText	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_MSG_DISCARDED	UNMAPPABLE MESSAGE DISCARDED
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.6 VCI_Login

Description

This Call Interface entry point is used by the end user application to login to Exchange Services. Before sending any application requests, user authorization must be acquired from the appropriate Exchange Service by calling VCI_Login for each desired Exchange Service. This is an asynchronous request. For Xervices that implement the Broadcast Extension (see *section 2.5.2*), authorization is required for subscriptions to broadcast streams as well.

VCI_Login requires registration of an application callback. The login callback is used for receipt of login and logout responses or to notify the end user application of exceptional cases (e.g., failure of Exchange Service).

The login response is received asynchronously via the registered callback. The status (*techComplCode*) of the application callback explicitly indicates receipt of login and logout responses.

The end user application may call VCI_Login multiple times to login several users. The login response returns a unique *loginID* for each successful login. The returned *loginID* must be specified for each subsequent call to VCI_Submit or VCI_Logout to authorize the specific user. In case of a Xervice using Broadcast Extension (see *section 2.5.2*), this also applies to VCI_Subscribe.

Syntax

```
void VCI_Login(
        ReqCntrlT           *reqControl,
        LoginReqDataT       *reqData,
        AppCallbackT        *callbackFunc,
        AppCntxtDataT       *callbackCntxtData,
        LoginRespDataT      *respData,
        StatusDataT         *statusDataGlobal
        );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	m	A
	loginID	r	-
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
reqData [1]	userID	m	A
	applVersion	m	A
	authorizationData	m	A
	authorizationDataLength	m	A
callbackFunc	(ref. to the application callback)	m	A
callbackCntxtData	custBlockSize	m	A
	custData	o	A
respData	n/a	r	-
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

1.Refer to Exchange application specific volumes for details.

Status Field Name	Value	Text
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS VCI_WARNING VCI_ERROR VCI_FATAL	
complCode, complText	n/a	
techComplCode, techCompltext	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_LOGIN_NUM_EXCEEDED	MAXIMUM NUMBER OF LOGINS EXCEEDED
	ELB_TECH_XERVICE_NOT_AVAILABLE	EXCHANGE SERVICE NOT AVAILABLE
	ELB_TECH_TOO_MANY_PENDING_REQUESTS	TOO MANY PENDING REQUESTS IN QUEUE
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.6.1 The Login Application Callback

The login application callback is used to receive login and logout responses and to notify the end user application of exceptions (e.g., Exchange Service unavailable). Please refer to *section 3.11* for details on the application callback function type, parameter, and completion codes. For login responses, the following data (*LoginRespDataT*) is returned by the application callback:

Parameter Name	Fields	Usage	Filled By
appRespData	funcResult	r	-
	loginID	ro	V

The status of the login and logout response or notification of exceptions is returned with the status data of the login application callback.

3.7 VCI_Logout

Description

This Call Interface entry point is used by the end user application to logout from Exchange Services.

VCI_Logout is an asynchronous request.

VCI_Logout performs a logout of a specific user login, identified by the loginID.

VALUES forwards the logout request to the specified Exchange Service. The response of a logout request is received asynchronously via the application callback registered with the corresponding login request. Receipt of logout responses is indicated explicitly through the status (*techComplCode*) of the application callback. Please refer to *section 3.11* and *section 3.6.1* for details on the login application callback.

Syntax

```
void VCI_Logout(
        ReqCntrlT               *reqControl,
        LogoutReqDataT          *reqData,
        LogoutRespDataT         *respData,
        StatusDataT             *statusDataGlobal
        );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	m	A
	loginID	m	A
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-

reqData	n/a	r	-
respData	n/a	r	-
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

Status Field Name	**Value**	**Text**
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode, complText	n/a	
techComplCode, techComplText	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_TOO_MANY_PENDING_REQUESTS	TOO MANY PENDING REQUEST IN QUEUE
	ELB_TECH_NOT_LOGGED_IN	USER NOT LOGGED IN
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.8 VCI_Submit

Description

This Call Interface entry point is used by end user applications to send processing requests to Exchange Services. The end user application specifies a request ID, request data, and an application callback. Associated with the callback, the application can specify custom context data which pointer is returned to the application when the callback is invoked. VALUES stores only the pointers to the requests and the custom context data. It does not store the actual data. VALUES forwards the request to the appropriate Exchange Service. The application response is routed back to the requester through the application-specified callback.

When calling VCI_Submit, the *loginID* must be specified to authenticate a specific user. The *loginID* must have been previously obtained through VCI_Login.

VCI_Submit is an asynchronous entry point.

Syntax

```
void VCI_Submit(
ReqCntrlT                *reqControl,
SubmitReqDataT           *reqData,
    AppCallbackT         *callbackFunc,
    AppCntxtDataT        *callbackCntxtData,
StatusDataT              *statusDataGlobal
);
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	m	A
	loginID	m	A
	appDescr	r	-
	reqID	m	A
	resubmitFlag	r	-
	resubmitNo	o[1]	A
reqData	appReqBlockSize	m	A
	appReq	m	A
callbackFunc	(reference to callback function)	m	A
callbackCntxtData	custBlockSize	m	A

	custData	o	A
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

1. Application request resubmission is supported by some Exchange applications. Please refer to the Exchange specific volumes for details.

Status Field Name	Values	Text
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode, complText	n/a	
techComplCode, techComplText	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_TOO_MANY_PENDING_REQUESTS	TOO MANY PENDING REQUESTS IN QUEUE
	ELB_TECH_NOT_LOGGED_IN	USER NOT LOGGED IN
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.8.1 The Submit Application Callback

The application callback is used to notify the end user application of responses and exceptions (e.g., GATE unavailable). Please refer to *section 3.11* for details on the application callback function type, parameter and completion codes.

Parameter Name	Fields	Usage	Filled By
appRespData	Refer to Exchange application specific volumes	-	-

The status of the request processing or notification of exception is returned with the status data of the submit application callback.

3.9 VCI_Subscribe

Description

This Call Interface entry point is used by the end user application to subscribe to data streams. Before subscribing to a data stream, the application must have successfully established a session (via VCI_Connect). For Xervices that implement the Broadcast Extension (see *section 2.5.2*), login authorization is required as well. On calling VCI_Subscribe, the application must identify the desired data stream and register an application callback. Associated with the callback, the application can specify custom context data which is returned to the application when the callback is invoked. VALUES stores only the pointers to the request and the custom context data.

VCI_Subscribe is an asynchronous request which can result in multiple types of responses: subscription response and subscription data. The response to a subscription request is received asynchronously via the registered application callback. Receipt of subscription responses is indicated explicitly through the status (*techComplCode*) of the application callback. A subscription is active only after receipt of a successful subscription response.

Data arriving through an active subscription is also passed to the end user application via the same registered application callback. The end user application can distinguish between received subscription data and received subscription/unsubscription responses through the status (techComplCode) of the application callback. Please refer to *section 3.11* for the specific techComplCode.

A special subscription subject is available that enables the transmission of Gap Notifications within a stream it has been used on. For this purpose, reqData.subject, reqData.subjectLength and reqData.applVersion have to be set to the special values SUBJECT_GAPINFO, strlen(SUBJECT_GAPINFO), and SUBJECT_GAPINFO_VERSION. Please

note that a wildcard subscription is not sufficient to receive Gap Messages. When receiving gap information in the callback registered by VCI_Subscribe, the subject is set to SUBJECT_GAPINFO. The subscription must be unsubscribed normally when Gap Messages are no longer required.

For recoverable broadcasts, usage of the Exchange application specific procedure (Broadcast Retransmission) should be preferred over the SUBJECT_GAPINFO mechanism.

Syntax

```
void VCI_Subscribe(
        ReqCntrlT           *reqControl,
        SubsReqDataT        *reqData,
        AppCallbackT        *callbackFunc,
        AppCntxtDataT       *callbackCntxtData,
        SubsRespDataT       *respData,
        StatusDataT         *statusDataGlobal );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	m	A
	loginID	m/r	A, only for Xervices with Broadcast Extension, see *section 2.5.2*
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
reqData	subsSubject	r	-
	applVersion[1]	m	A
	streamType	m	A
	subjectLength	m	A
	subject[2]	m	A
	userID	r	-

	authorizationDataLength	r	-
	authorizationData	r	-
callbackFunc	(reference to callback function)	m	A
callbackCntxtData	custBlockSize	m	A
	custData	o	A
respData	n/a	r	-
statusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	TechComplCode	ro	V
	TechComplText	ro	V

1. For subscription to the stream's Gap Messages, set applVersion = SUBJECT_GAPINFO_VERSION, subject = SUBJECT_GAPINFO, and subjectLength = strlen(SUBJECT_GAPINFO).
2. The structure of subject is defined in the Exchange specific volumes with each subscription request.

Status Field Name	Value	Text
ComplSeverity	n/a	
TechComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode, complText	n/a	
techComplCode, techComplText	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED

ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
ELB_TECH_TOO_MANY_PENDING_ REQUESTS	TOO MANY PENDING REQUESTS IN QUEUE
ELB_TECH_NOT_LOGGED_IN	USER NOT LOGGED IN (only for Xervices that implement Broadcast Extension, see *section 2.5.2*
ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.9.1 The Subscription Application Callback

The subscription application callback is used to receive subscription responses. Please refer to *section 3.11* for details on the application callback function type, parameter and completion codes.

For subscription responses, the following data (*SubsRespDataT*) is returned by the application callback.

Parameter Name	Fields	Usage	Filled By
AppRespData	subsID	ro	V

The status of the subscription response or notification of exceptions is returned with the status data of the subscription application callback.

Broadcasts may occur at high frequency, depending on which subscriptions are active. To make sure that an application can receive these broadcasts fast enough to avoid the VMQ being overrun, implementing lean subscription callbacks should be a primary design goal. If processing the broadcast data can be expected to take a considerable amount of time, it should be done in a way that does not slow down the reception of further broadcasts. In case of a VMQ overflow, GATE drops the application's VALUES session, and the application is being disconnected.

3.10 VCI_Unsubscribe

Description

This Call Interface is used by the end user application to end subscription to the specified data stream. Ending a subscription results in turning off the data stream and removing the broadcast callback reference maintained by VALUES.

VCI_Unsubscribe is an asynchronous request.

The response of an unsubscription request is received via the application callback registered with the corresponding subscription request. Receipt of unsubscription responses is indicated explicitly through the status (*techComplCode*) of the application callback.

Please refer to *section 3.9.1* for details on the subscription application callback.

Syntax

```
void VCI_Unsubscribe(
        ReqCntrlT           *reqControl,
        UnsubsReqDataT      *reqData,
        UnsubsRespDataT     *respData,
        StatusDataT         *statusDataGlobal
        );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	m	A
	connectionID	m	A
	dbApplID	r	-
	loginID	r	-
	appDescr	r	-
	reqID	r	-
	resubmitFlag	r	-
	resubmitNo	r	-
reqData	subsID	m	A
respData	n/a	r	-
StatusDataGlobal	complSeverity	r	-
	complCode	r	-
	complText	r	-
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

Status Field Name	Value	Text
complSeverity	n/a	
techComplSeverity	VCI_SUCCESS	
	VCI_WARNING	
	VCI_ERROR	
	VCI_FATAL	
complCode, complText	n/a	
techComplCode, techComplText	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_INVALID_PARAMETER	INVALID PARAMETER PASSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
	ELB_TECH_NOT_CONNECTED	APPLICATION NOT CONNECTED
	ELB_TECH_TOO_MANY_PENDING_REQUESTS	TOO MANY PENDING REQUESTS IN QUEUE
	ELB_TECH_NOT_SUBSCRIBED	STREAM NOT SUBSCRIBED
	ELB_TECH_NOT_REENTRANT	VALUES CALLS ARE NOT REENTRANT

3.11 Application Callback Function Type

Type Name: AppCallbackT

Description: Callbacks of this type are implemented in the end user application and are invoked by VALUES to perform asynchronous processing in response to application requests (including login, logout, subscribe and unsubscribe), subscriptions, and exceptions. All callbacks have the same interface and can be registered through the entry points VCI_Connect, VCI_Login, VCI_Submit, and VCI_Subscribe.

Associated with a registered callback, the application can specify custom context data which pointer is returned to the application when the callback is invoked. VALUES stores only the pointers to the request and the custom context data. The actual data is not stored by VALUES. Both fields have to be de-allocated by the end user application. Application callbacks are called synchronously by VCI_Dispatch.

All data structures are allocated and populated by the application. The statusData structure passes status about the processing of the request, subscription, or exception to the end user application.

The data structure pointed at by the parameter appData contains application response and application request data. The pointer to the original application request data is passed back to the application.

Note: It is not allowed to call any VALUES API entry points from within an application callback. Calling any VCI entry point will fail with ELB_TECH_NOT_REENTRANT when used inside a callback invoked by VCI_Dispatch.

Syntax:

```
typedef void (AppCallbackT) (
        ReqCntrlT           *reqControl,
        CallBkAppDataT      *appData,
        AppCntxtDataT       *callbackCntxtData,
        StatusDataT         *statusDataGlobal
        );
```

Parameter Name	Fields	Usage	Filled By
reqControl	VCIver	r	-
	connectionID	r	-
	dbApplID [1]	ro	V
	loginID [1]	ro	V
	appDescr	r	-
	reqID	ro	V
	resubmitFlag [2]	ro	V
	resubmitNo [2]	ro	V
AppData	appRespBlockSize	ro	V
	appRespData	ro	V
	subsID	ro	V (used for subscription callbacks only)
	subject	r	-
	appReqBlockSize	ro	V (used for request callbacks only)
	appReqData	ro	V
	applVersion	ro	V (except VCI_Connect callback)
	streamType	ro	V (used for subscription callbacks only)
	brcSubject[3]	ro	V (used for subscription callbacks only)
CallbackCntxtData	custBlockSize	ro	V
	custData	ro	V
StatusDataGlobal	complSeverity	ro	V
	complCode	ro	V
	complText	ro	V
	techComplSeverity	ro	V
	techComplCode	ro	V
	techComplText	ro	V

1. Always set, except in case of connection callback with techComplCode: ELB_TECH_OK, ELB_TECH_TECHSRVC_NOT_AVAILABLE, ELB_TECH_REQ_UNSUCCESSFUL
2. Only set in case of login callback (except for login response), in connection callback with techComplCode: ELB_TECH_LOGGED_OUT and in submission callback
3. The structure of brcSubject is defined in the Exchange specific volumes with each subscription request.

Status Field Name	Value
ComplSeverity	VCI_SUCCESS
TechComplSeverity	VCI_WARNING
	VCI_ERROR
	VCI_FATAL
complCode,complText	These fields return Exchange application specific completion status. Please refer to the Exchange application specific volumes for a detailed list of completion codes generated by each Exchange application.

Connection Callback		
Triggering VCI entry point	**techComplCode**	**techComplText**
VCI_Dispatch (error reading message sent by GATE)	ELB_TECH_TECHSRVC_NOT_ AVAILABLE	TECHNICAL SERVICES NOT AVAILABLE
VCI_Dispatch (Exchange Service notification)	ELB_TECH_XERVICE_NOT_ AVAILABLE	EXCHANGE SERVICE NOT AVAILABLE
	ELB_TECH_XERVICE_AVAILABLE	EXCHANGE SERVICE AVAILABLE
	ELB_TECH_NONTRANSPARENT_ FAILOVER	NONTRANSPARENT FAILOVER
VCI_Dispatch (disconnect state caused by callback invocation)	ELB_TECH_TECHSRVC_NOT_ AVAILABLE	TECHNICAL SERVICES NOT AVAILABLE
VCI_Dispatch (response of VCI_Logout request)	ELB_TECH_LOGGED_OUT	USER LOGGED OUT SUCCESSFULLY
VCI_Disconnect (syncronous)	ELB_TECH_OK [1]	SUCCESSFUL COMPLETION
	ELB_TECH_REQ_ UNSUCCESSFUL[2]	REQUEST NOT SUCCESSFULLY PROCESSED

Table 3.3 - Connection Callback

1. Indicates successful disconnection
2. Indicates failed disconnection

Login Callback		
Triggering VCI entry point	**techComplCode**	**techComplText**
VCI_Dispatch (response of VCI_Login request)	ELB_TECH_LOGGED_IN [1]	USER LOGGED IN SUCCESSFULLY
	ELB_TECH_INTERNAL_ERROR [2]	INTERNAL ERROR OCCURRED
	ELB_TECH_REQ_ UNSUCCESSFUL[3]	REQUEST NOT SUCCESSFULLY PROCESSED
VCI_Dispatch (error reading message sent by GATE)	ELB_TECH_TECHSRVC_NOT_ AVAILABLE	TECHNICAL SERVICES NOT AVAILABLE
	ELB_TECH_PENDING_REQUEST_ DELETED [4]	PENDING REQUEST DELETED

Table 3.4 - Login Callback

1. Indicates receipt of a successful login response containing login ID
2. Indicates internal error while processing of login request
3. Login not allowed
4. Indicates unconfirmed login request

Login Callback (Logout)		
Triggering VCI entry point	**techComplCode**	**techComplText**
VCI_Dispatch (Exchange Service notification)	ELB_TECH_XERVICE_NOT_AVAILABLE	EXCHANGE SERVICE NOT AVAILABLE
	ELB_TECH_NONTRANSPARENT_FAILOVER	NONTRANSPARENT FAILOVER
	ELB_TECH_PENDING_REQUEST_DELETED[1]	PENDING REQUEST DELETED
VCI_Dispatch (response of VCI_Logout request)	ELB_TECH_LOGGED_OUT[2]	USER LOGGED OUT SUCCESSFULLY
	ELB_TECH_TECHSRVC_NOT_AVAILABLE	TECHNICAL SERVICES NOT AVAILABLE
VCI_Disconnect	ELB_TECH_XERVICE_NOT_AVAILABLE	EXCHANGE SERVICE NOT AVAILABLE
	ELB_TECH_PENDING_REQUEST_DELETED[1]	PENDING REQUEST DELETED

Table 3.5 - Login Callback (Logout)

1. Indicates unconfirmed login/logout request
2. Indicates receipt of a successful logout response

Subscription Callback		
Triggering VCI entry point	**techComplCode**	**techComplText**
VCI_Dispatch (response of VCI_Subscribe request)	ELB_TECH_SUBSCRIBED [1]	STREAM SUBSCRIBED
	ELB_TECH_INTERNAL_ERROR [2]	INTERNAL ERROR OCCURRED
	ELB_TECH_OK [3]	SUCCESSFUL COMPLETION
	ELB_TECH_RQ_UNSUCCESSUL	REQUEST NOT SUCCESSFULLY PROCESSED
VCI_Dispatch (receipt of broadcast messages)	ELB_TECH_OK [3]	SUCCESSFUL COMPLETION
VCI_Dispatch (error reading message sent by GATE)	ELB_TECH_SUBSCRIPTION_ DELETED	SUBSCRIPTION DELETED
	ELB_TECH_PENDING_REQUEST_ DELETED [4]	PENDING REQUEST DELETED
VCI_Dispatch (response of VCI_Unsubscribe request)	ELB_TECH_UNSUBSCRIBED [5]	STREAM UNSUBSCRIBED
	ELB_TECH_INTERNAL_ERROR [2]	INTERNAL ERROR OCCURRED
VCI_Dispatch (response of VCI_Logout, only for Xervices with Broadcast Extension, see *section 2.5.2*)	ELB_TECH_SUBSCRIPTION_ DELETED	SUBSCRIPTION DELETED
VCI_Disconnect	ELB_TECH_SUBSCRIPTION_ DELETED	SUBSCRIPTION DELETED
	ELB_TECH_PENDING_REQUEST_ DELETED [4]	PENDING REQUEST DELETED

Table 3.6 - Subscription Callback

1. Indicates receipt of a successful subscription response
2. Indicates internal error while processing the subscription request
3. Indicates successful receipt of a broadcast
4. Indicates unconfirmed subscription/unsubscription request
5. Indicates receipt of a successful unsubscription response

Submit Callback		
Triggering VCI entry point	**techComplCode**	**techComplText**
VCI_Dispatch (response of VCI_Submit request)	ELB_TECH_OK	SUCCESSFUL COMPLETION
	ELB_TECH_REQ_UNSUCESSFUL	REQUEST NOT SUCCESSFULLY PROCESSED
	ELB_TECH_INTERNAL_ERROR	INTERNAL ERROR OCCURRED
VCI_Dispatch (error reading message sent by GATE)	ELB_TECH_PENDING_REQUEST_DELETED	PENDING REQUEST DELETED
VCI_Dispatch (Exchange Service notification)	ELB_TECH_PENDING_REQUEST_DELETED	PENDING REQUEST DELETED
VCI_Dispatch (response of VCI_Logout request)	ELB_TECH_PENDING_REQUEST_DELETED	PENDING REQUEST DELETED
VCI_Disconnect	ELB_TECH_PENDING_REQUEST_DELETED	PENDING REQUEST DELETED

Table 3.7 - Submit Callback

4 VALUES API Usage Examples

4.1 Overview

This section gives a collection of code fragments that describe the usage of the VALUES API Call Interface entry points. The programming example consists of code fragments and explanatory text. The code fragments include calls to the VALUES API, parameter specification, memory allocation, and examples of application callbacks.

The section is split into subsections outlining the path from initiation of a session, logging into Exchange applications, end user application requests/responses including a dispatch loop, logging out of Exchange applications and termination of a session.

The code fragments in this section are intended to demonstrate how certain tasks that are involved with VALUES programming can be implemented in an end user application. They do not make up a complete example program. It is not recommended to use them as a template for VALUES-based Front End applications without adaptions.

The examples assume use of the Sun Solaris platform.

4.2 Initiating a VALUES Connection

The VCI_Connect entry point has to be used to connect the end user application to VALUES. After a successful VCI_Connect, the end user application can subscribe to data streams and receive broadcasts. It can also login to Exchange services and then send application requests and receive responses.

The following list describes how the end user application can allocate and de-allocate memory when using VCI_Connect:

- create connection handler
- allocate memory for request data
- allocate memory for context data
- call VCI_Connect (synchronous)
- depending on VCI_Connect's techComplCode
- if ELB_TECH_OK *(connected)*
- de-allocate request data's memory
- else *(connect failed)*
- de-allocate request data's memory
- de-allocate context data's memory
- delete connection handler
- wait to be informed about state changes using the connection callback
- depending on connection callback's techComplCode:
- if ELB_TECH_TECHSRVC_NOT_AVAILABLE *(disconnected)*
- de-allocate context data's memory

- delete connection handler
- no action for all other completion codes
- call VCI_Disconnect (synchronous)
- if VCI_Disconnect's completion code neither ELB_TECH_INVALID_PARAMETER nor ELB_TECH_NOT_CONNECTED *(disconnected)*
- de-allocate context data's memory
- delete connection handler

Code example for VCI_Connect:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <poll.h>               // The library is available on the SUN platform.
                                // The poll function is here used to poll
                                // the VMQ (VALUES Message Queue).

// VALUES specific include files
#include "Values.h"             // This file declares all types and functions
                                // of the VCI (VALUES Call Interface).
#include "elbcodetech.h"        // This file declares the technical completion codes

// Eurex include files
#include "XEUR_elbcode.h"       // This file declares the functional completion codes
#include "DRIV_data_types.hxx"  // This file contains constants that specify sizes
                                // and valid values of data fields.
#include "DRIV_app_rid.h"       // The file contains constant application request IDs.
#include "DRIV_login.h"         // This file contains a structure definition of Eurex
                                // authorization data.
#include "EntSLegOrdr.hxx"      // Include the application request structure definitions
                                // needed in the Enter Order submission below.

// Xetra include files
#include "app_rid.h"
#include "xbrdcast.h"
#include "subjectXetra.h"
#include "xetra_login.h"
#include "vld_val.h"

// global structs and variables
// most of them are used to allow communication of callback functions with the main
// program, or to make sure variables that have to be accessible in both callbacks
// and main program are in scope in both environments.
// Real-life applications should use custom context data structs and malloc()'d
// memory to handle this.
char    myContextData [100];    // The connectCallback receives a pointer to
                                // myContextData in callbackCntextData
                                // from VALUES and needs to access this data.
                                // The data has to be valid until the session
                                // is terminated

LoginReqDataT*   LoginData;     // login request data block
SubmitReqDataT*  SubmitData;    // submit request data block
```

```
optEntSLegOrdrRequestT   my_appl_request_struct;
                                 // The responseCallback
                                 // receives a pointer appReq =
                                 // &my_appl_request_struct in order to
                                 // identify the request that triggered the response.
                                 // Therefore, my_appl_request_struct
                                 // must be stored, for example, by declaring it as
                                 // a global variable. The data has to be valid until
                                 // the response is handled.

int callbackLoginId;         // receives the loginID within the loginCallback function
int subsId;                  // broadcast subscription ID

#define XERVICE_YET_UNKNOWN       (-1)
int xetraDbApplId = XERVICE_YET_UNKNOWN;      // Xetra Transaction Xervice Application Id
                                              // to be stored by the Connect callback
int eurexDbApplId = XERVICE_YET_UNKNOWN;      // Eurex Transaction Xervice Application Id
                                              // to be stored by the Connect callback

int main (int argc, char** argv)
{
    // Local variables
    int     fd;         // fd is the file descriptor of the VMQ.
    char    prodMode;   // Application can use prodMode to inform
                        // the user about the production mode.

    // Define variables in VCI-format:
    ReqCntrlT           reqControl1;                 // request control records,
    ReqCntrlT           reqControl2;                 //   one for each user
    CnctReqDataT*       cnctreqData;                 // connection request data block
    SubmitReqDataT*     submitreqData;               // submit request data block
    DiscnctReqDataT*    discnctreqData;              // disconnect request data block
    CnctRespDataT       cnctrespData;                // connection response data block
    StatusDataT         statusData;                  // VCI function completion status record
    AppCntxtDataT*      callbackContextData;         // App. Callback context data block record
    DRIVLoginAuthorizationDataT*     DRIV_AuthData;  // Authorization structure for Eurex login
    XetraLoginAuthorizationDataT*    xetraAuthData;  // Authorization structure for Xetra login

    strcpy(reqControl1.VCIver, CVN_012);          // Set VALUES version for compatibility checking;

    // allocate memory for cnctreqData (this memory can be free()d after
    // returning from VCI_Connect)
    cnctreqData = (CnctReqDataT*) malloc( sizeof(CnctReqDataT) );

    // allocate context data to be used througout the VALUES session
    callbackContextData = (AppCntxtDataT*) malloc( sizeof(AppCntxtDataT ));

    // fill the fields of the cnctreqData-structure
    // the userId and password should come from a user or a file and not be hardcoded
    strcpy(cnctreqData->userID, "myuser");        // FE operating system userID for VALUES or
                                                  // DB-application
    strcpy(cnctreqData->password, "mypassword");  // FE operating system password for userID

    // fill the fields of the callbackContextData-structure
    // This example uses no context data
    callbackContextData->custBlockSize = 0;
    callbackContextData->custData = NULL;
```

```
    // Call VCI_Connect to connect the application to VALUES.
    // This is needed to perform a login to the Back End systems.
    VCI_Connect(     &reqControl1,
                     cnctreqData,
                     connectCallback,
                     callbackContextData,
                     &cnctrespData,
                     &statusData
                     );

    // Assess whether or not the call succeeded.
    ret_status_handling( &statusData );

    ///////////////////////////////////////////////////////////////
    // We have return-data received from VCI_Connect.
    // Now save response-data for use in later VALUES processing.
    ///////////////////////////////////////////////////////////////
    // The Production Mode is stored.
    prodMode = cnctrespData.prodMode;
    // The file descriptor to the VMQ is stored.
    fd = atoi(cnctrespData.VMQname);
    // The connection ID is stored to identify the session in
    // all future interface calls.
    reqControl1.connectionID = cnctrespData.connectionID;

    // In the function below we poll the VMQ and call VCI_Dispatch upon event.
    // This call is performed to receive the asynchronously sent data
    // about the availability of Xervices
    poll_and_dispatch ( &reqControl1, fd );

    // Continue here with application specific processing
}
```

4.3 Application Dispatch upon Event Notification

The VCI_Dispatch entry point has to be used by the end user application to receive responses, broadcasts and exceptions. VCI_Dispatch will read response data from the VMQ and pass it on to the appropriate application response callback.

Code example for polling the VMQ and calling VCI_Dispatch:

```
void poll_and_dispatch (      ReqCntrlT    *reqControl,     // request control record
                              int          fd               // file descriptor of VMQ
                              )
{
    StatusDataT      statusData;  // VCI function completion status record
    struct pollfd    pfd;         // This variable is used to store the polling-output of the VMQ.

    // Poll the VMQ.
    // It is important to NOT read from the VMQ.
    // The actual read is performed by VCI_Dispatch.

    pfd.fd       = fd;
    pfd.events   = 0;
    pfd.events   = pfd.events | POLLIN;
```

```
    pfd.events   = pfd.events | POLLRDNORM;
    pfd.events   = pfd.events | POLLRDBAND;
    pfd.events   = pfd.events | POLLPRI;
    pfd.events   = pfd.events | POLLERR;

    if (poll(&pfd, 1, -1) < 0)        // Blocks until VMQ event occurs.
                                      // Use select() with other platforms.
    {
        printf("\n Poll failed");
        break;
    }

    // If a message is available, then call VCI_Dispatch.
    if ( (pfd.revents & POLLERR) != 1 )
    {
        // Call VCI_Dispatch to read the response data and call VCI_Dispatch upon event.
        VCI_Dispatch(    reqControl,
                         &statusData
                         );

        // Assess whether or not the call succeeded.
        ret_status_handling(&statusData);
    }
    else
    {
        printf("\n Error on asynchronous channel");
    }

    return;
}
```

4.4 Receiving Connection Events

The following code fragment gives an example on how a callback can be implemented in the end user application to perform asynchronous processing of connection events.

Code example for a connection callback:

```
void connectCallback (    ReqCntrlT    *reqControl,
                          CallBkAppDataT*appData,
                          AppCntxtDataT*callBackCntxtData,
                          StatusDataT  *statusDataGlobal
                          )
{
    //used to decode "Xervice Available" and /"Xervice not available" - messages
    XerviceInfoT*    pxinfo;

    printf( "\n**************************************** \n");
    printf( "\tConnection Callback \n");
    printf( "**************************************** \n");

    switch( statusDataGlobal->techComplCode )
    {
        // This function does not handle all completion codes that may occur..
        case ELB_TECH_OK:
            printf( "Disconnection request Successfully Processed\n" );
```

```
            break;
        case ELB_TECH_REQ_UNSUCCESSFUL:
            // In this exception case the application must determine,
            // based on the detailed context
            // (exception detail, request data, trading phase etc.) whether the request
            // must be resubmitted or not
            printf( "\n%s\n", statusDataGlobal->techComplText );
            // The functional completion code provides detailed information:
            printf( "%s\n", statusDataGlobal->complText );
            break;
        case ELB_TECH_LOGGED_OUT:
            // the application has logged out of the Xervice.
            // Usually, this requires no action by the
            // connect callback, since the corresponding login callback gets invoked as well.
            break;
        case ELB_TECH_XERVICE_AVAILABLE:
            // Extract the Xervice Id numbers of Xervices we are interested in
            // To use multi-exchange capability, an application would store any
            // Xervice Id that comes with a Xervice Class it is ready to handle,
            // as well as additional information from XerviceInfoT. The example
            // code does not use multi-exchange, but pre-defined exchanges.
            pxinfo = (XerviceInfoT*) appData->appRespData;
            // We are specifically interested in the dbApplID of the Transaction Xervice offered
            // by Xetra Frankfurt (MIC is "XETR")
            if (pxinfo->applClass == XETRA_TXN_XCLASS &&
                memcmp(pxinfo->exchApplId, "X" EXCH_ID_COD_XETRA, 4) == 0)
            {
                xetraDbApplId = reqControl->dbApplID;
                break;
            }
            // similar for Eurex
            if (pxinfo->applClass == EUREX_TXN_XCLASS &&
                memcmp(pxinfo->exchApplId, EXCH_APPL_ID_EUREX, 4) == 0)
            {
                eurexDbApplId = reqControl->dbApplID;
            }
            break;
        case ELB_TECH_XERVICE_NOT_AVAILABLE:
            // on application startup, this event is sent for each Xervice that is configured
            // on the MISS, but not currently available. Thus, we can collect XerviceInfo
            // from here as well.
            pxinfo = (XerviceInfoT*) appData->appRespData;
            // We are specifically interested in the dbApplID of the Transaction Xervice offered
            // by Xetra Frankfurt (MIC is "XETR")
            if (pxinfo->applClass == XETRA_TXN_XCLASS &&
                memcmp(pxinfo->exchApplId, "X" EXCH_ID_COD_XETRA, 4) == 0)
            {
                    xetraDbApplId = reqControl->dbApplID;
                    break;
            }
            // similar for Eurex
            if ( pxinfo->applClass == EUREX_TXN_XCLASS &&
                 memcmp(pxinfo->exchApplId, EXCH_APPL_ID_EUREX, 4) == 0)
            {
                    eurexDbApplId = reqControl->dbApplID;
            }
            break;
            // The application can wait for the exchange to become available
        case ELB_TECH_INTERNAL_ERROR:
```

```
            // The application should disconnect and try to restart. This error is mostly caused by
            // MISS/WS configuration or setup problems.
        case ELB_TECH_TECHSRVC_NOT_AVAILABLE:
            // The application will be disconnected from VALUES at this point.
            // It may re-connect when the
            // technical services have been restarted (by the system operator if needed)
            printf( "\n%s\n", statusDataGlobal->techComplText );
            break;
        default:
            printf( "\nUnknown Error! Connection Id: %d\n", reqControl->connectionID );
            if (callBackCntxtData)
            {
                printf("callBackCntxtData.custData = %s\n", (char*)callBackCntxtData->custData);
            }
            printf("Application Version = %d\n", appData->applVersion);
            break;
    }
    return;
}
```

4.5 Logging on to Exchange Services

The VCI_Login entry point has to be used to login to Exchange services. Before sending any application request, the user must get authorization from the Exchange service. For Xervices that implement the Broadcast Extension (see *section 2.5.2*), authorization is also required for broadcast subscription. Multiple users can login to an Exchange services using the same VALUES session.

The following list describes how the end user application can allocate and de-allocate memory when using the VCI_Login and VCI_Logout:

- create login handler
- allocate memory for request data
- allocate memory for context data
- mark login handler as “pending”
- call VCI_Login (asynchronous)
- if VCI_Login’s techComplCode not ELB_TECH_OK *(login failed)*
- de-allocate context data’s memory
- de-allocate request data’s memory
- delete login handler
- wait to be informed about state changes using the login callback
- depending on login callback’s techComplCode
- if ELB_TECH_LOGGED_IN *(logged in)*
- mark login handler as “logged in”
- de-allocate request data’s memory
- if either ELB_TECH_REQ_UNSUCCESSFUL or ELB_TECH_INTERNAL_ERROR *(error while processing login request)*

- de-allocate context data's memory
- de-allocate request data's memory
- delete login handler
- if ELB_TECH_NONTRANSPARENT_FAILOVER
- no action
- if either ELB_TECH_LOGGED_OUT or ELB_TECH_TECHSRVC_NOT_AVAILABLE *(logged out)*
- de-allocate context data's memory
- delete login handler
- if ELB_TECH_PENDING_REQUEST_DELETED
- if login handler is "pending" *(error while processing login request)*
- de-allocate context data's memory
- de-allocate request data's memory
- delete login handler
- else *(error while processing logout request)*
- no action
- if ELB_TECH_XERVICE_NOT_AVAILABLE
- if login handler "logged in" *(logged out)*
- de-allocate context data's memory
- delete login handler
- else *(error while processing login request)*
- de-allocate request data's memory
- de-allocate context data's memory
- delete login handler
- call VCI_Logout (asynchronous)
- no action

Code example for VCI__Login:

```
// Duplicate request control record for use with second user.
reqControl2 = reqControl1;

// Allocate memory for LoginData
//(this memory has to be deallocated in the login/out callback function).
LoginData = (LoginReqDataT*) malloc( sizeof(LoginReqDataT) );

// Fill the fields of the reqData-structure for the first user.
// Assign application version as defined in DRIV_data_types.hxx.
LoginData->applVersion = XEUR_AVN_090;
```

```
// The userId and password should come from a user or a file and not be
// hardcoded as they are here.
memcpy(LoginData->userID, "MEMBRTRD001", LOGIN_MAX_USERID);
DRIV_AuthData = (DRIVLoginAuthorizationDataT*) malloc(sizeof(DRIVLoginAuthorizationDataT));
memcpy(DRIV_AuthData->password, "MYPASSWD", DRIV_LOGIN_MAX_PWDID);      // password for userID

// Here the application ID assigned by the exchange should be used.
memcpy(DRIV_AuthData->applicationId, "EUREXXXXXXXXXXXX", DRIV_LOGIN_APPLICATION_ID );
LoginData->authorizationData          = (void*) DRIV_AuthData;
LoginData->authorizationDataLength    = sizeof(DRIVLoginAuthorizationDataT);

strcpy(reqControl1.appDescr, "Eurex");            // This is an optional description.
// check if the Xervice was announced as available: (this happens inside the connectCallback)
assert( eurexDbApplId != XERVICE_YET_UNKNOWN );
reqControl1.dbApplID = eurexDbApplId;             // Logon will be done for Eurex.

// Call VCI_Login for the first user. This gives the user the authorization
// from the exchange Back End to submit application requests.
VCI_Login(   &reqControl1,
             LoginData,
             loginCallback,
             NULL,
             NULL,
             &statusData
             );

// Assess whether or not the call succeeded.
ret_status_handling(&statusData);

// In this function we poll the VMQ and call VCI_Dispatch upon event.
poll_and_dispatch ( &reqControl1, fd );

// The loginID has been received in loginCallback.
reqControl1.loginID = callbackLoginId;

// Allocate memory for LoginData
// (this memory has to be deallocated in the login/out callback function).
LoginData = (LoginReqDataT*) malloc(sizeof(LoginReqDataT));
// Fill the fields of the reqData-structure for the second user.
// The userId and password should come from a user or a file and not be
// hardcoded as they are here.
memcpy(LoginData->userID, "MEMBRTRD002", LOGIN_MAX_USERID);
xetraAuthData = (XetraLoginAuthorizationDataT*) malloc(sizeof(XetraLoginAuthorizationDataT));
memcpy(xetraAuthData->password, "MYPASSWD", XETRA_LOGIN_MAX_PWDID);        // password for userID
LoginData->authorizationData = (void*) xetraAuthData;
LoginData->authorizationDataLength = sizeof(XetraLoginAuthorizationDataT);

strcpy(reqControl2.appDescr, "Xetra");        // This is an optional description.
// check if the Xervice was announced as available:
assert( xetraDbApplId != XERVICE_YET_UNKNOWN );
reqControl2.dbApplID = xetraDbApplId;         // Logon will be done for Xetra.

// Call VCI_Login for the second user.
VCI_Login(   &reqControl2,
             LoginData,
             loginCallback,
             NULL,
             NULL,
```

```
                &statusData
                );

    // Assess whether or not the call succeeded.
    ret_status_handling(&statusData);

    // In this function we poll the VMQ and call VCI_Dispatch upon event.
    poll_and_dispatch ( &reqControl2, fd );

    // The loginID has been received in loginCallback.
    reqControl2.loginID = callbackLoginId;
```

4.6 Receiving Login Responses

The following code fragment shows how callbacks are implemented in the end user application to perform asynchronous processing in response to end user login requests.

Code example for a login callback:

```
extern LoginReqDataT*    LoginData;           // login request data block

void loginCallback (     ReqCntrlT        *reqControl,
                         CallBkAppDataT   *appData,
                         AppCntxtDataT    *callBackCntxtData,
                         StatusDataT      *statusDataGlobal
                         )
{
    // cast the void pointer to it's real type
    LoginRespDataT *respData = (LoginRespDataT*) appData->appRespData;

    printf( "\n**************************************** \n");
    printf( "\tLogin/out Callback \n");
    printf( "**************************************** \n");

    switch( statusDataGlobal->techComplCode )
    {
        // Here are only some of all the completion codes in elbcodetech.h handled.
        case ELB_TECH_LOGGED_IN:
            printf( "Logged In" );
            printf( "\n LOGIN ID: %ld\n", respData->loginID );
            // save the loginID just received into the global variable
            callbackLoginId = respData->loginID;
            break;
        case ELB_TECH_LOGGED_OUT:
            free(LoginData);
            printf( "Logged Out" );
            break;
        case ELB_TECH_REQ_UNSUCCESSFUL:
            printf( "\n%s\n", statusDataGlobal->techComplText );
            // The functional completion code provides detailed information:
            printf( "%s\n", statusDataGlobal->complText );
            break;
        case ELB_TECH_XERVICE_AVAILABLE:
        case ELB_TECH_XERVICE_NOT_AVAILABLE:
        case ELB_TECH_TECHSRVC_NOT_AVAILABLE:
```

```
        case ELB_TECH_NONTRANSPARENT_FAILOVER:
            printf( "\n%s\n", statusDataGlobal->techComplText );
            break;
        default:
            printf( "\n%s\n", statusDataGlobal->techComplText );
            printf( "Connection Id: %d\n", reqControl->connectionID );
            if (callBackCntxtData)
            {
                printf("callBackCntxtData.custData = %s\n", (char*)callBackCntxtData->custData);
            }
            break;
    }
    return;
}
```

4.7 Submitting Application Requests

The VCI_Submit Call Interface entry point has to be used by the end user application to send processing requests to the exchange application. The user has to specify a request code, request data and an application callback function.

The following list describes how the end user application can allocate and de-allocate memory when using the VCI_Submit:

- create submission handler
- allocate memory for context data
- allocate memory for request data
- call VCI_Submit (asynchronous)
- if VCI_Submit's techComplCode not ELB_TECH_OK *(submit failed)*
- de-allocate context data's memory
- de-allocate request data's memory
- delete submission handler
- wait to be informed about state changes (response) using the submission callback
- depending on submission callback's techComplCode
- for all completion codes
- de-allocate context data's memory
- de-allocate request data's memory
- delete submission handler

Code example for VCI_Submit:

```
    // The filling of the application request structure should be done at this point.
    // In general these fields are not hardcoded, but entered by a user.
    // This is an example of an Eurex stock option order entry.
    memcpy(my_appl_request_struct.header.exchApplId, EXCH_APPL_ID_EUREX, EXCH_APPL_ID_LEN);
    memcpy(my_appl_request_struct.header.prodLine, PROD_LINE_OPTION, PROD_LINE_LEN);
```

```
memset(my_appl_request_struct.header.membExchIdCodOboMS, EXCH_CONST_SPACE,
       MEMB_EXCH_ID_COD_OBO_MS_LEN);

my_appl_request_struct.extension.acctTypCod = ACCT_TYP_COD_AGENT;
my_appl_request_struct.extension.acctTypNo  = ACCT_TYP_NO_ONE;
memcpy(my_appl_request_struct.extension.txtGrp.cust, "My Customer ", CUST_LEN);
memcpy(my_appl_request_struct.extension.txtGrp.userOrdrNum, "123456789012",
        USER_ORDR_NUM_LEN);
memcpy(my_appl_request_struct.extension.txtGrp.text, "FreeFormText", TEXT_LEN);
memcpy(my_appl_request_struct.extension.membClgIdCod, "CLGMB", MEMB_CLG_ID_COD_LEN);
my_appl_request_struct.extension.prcRsblChkInd  = EXCH_CONST_NO;

my_appl_request_struct.basic.buyCod = EXCH_CONST_BUY;
memcpy(my_appl_request_struct.basic.optCntrIdGrp.prodId, "ODAX", PROD_ID_LEN);
my_appl_request_struct.basic.optCntrIdGrp.cntrClasCod = CNTR_CLAS_COD_CALL;
memcpy(my_appl_request_struct.basic.optCntrIdGrp.cntrExpMthDat, "06", CNTR_EXP_MTH_DAT_LEN);
memcpy(my_appl_request_struct.basic.optCntrIdGrp.cntrExpYrDat, "2006", CNTR_EXP_YR_DAT_LEN);
memcpy(my_appl_request_struct.basic.optCntrIdGrp.cntrExerPrc, "0004550", CNTR_EXER_PRC_LEN);
my_appl_request_struct.basic.optCntrIdGrp.cntrVersNo = CNTR_VERS_NO_ZERO;
memset(my_appl_request_struct.basic.trdrIdGrp.partSubGrpCod, EXCH_CONST_SPACE,
      PART_SUB_GRP_COD_LEN);
memset(my_appl_request_struct.basic.trdrIdGrp.partNo, EXCH_CONST_SPACE, PART_NO_LEN);
memcpy(my_appl_request_struct.basic.ordrQty, "+000000000177", DRIV_ORDR_QTY_LEN);
memcpy(my_appl_request_struct.basic.ordrExePrc, "+0000000002250", DRIV_ORDR_EXE_PRC_LEN);
my_appl_request_struct.basic.ordrResCod = EXCH_CONST_SPACE;
memcpy(my_appl_request_struct.basic.ordrExpDat, "20060630", ORDR_EXP_DAT_LEN);
my_appl_request_struct.basic.opnClsCod = OPN_CLS_COD_OPEN;

// Set the type of the request. The definition is in file "DRIV_app_rid.h".
reqControl1.reqID = DRIV_ENTER_SINGLE_LEG_ORDER_RID;

// allocate memory for callbackContextData
callbackContextData = (AppCntxtDataT*) malloc(sizeof(AppCntxtDataT));

// With SubmitData (global) we actually pass the application request data
// to the VALUES Call Interface.
SubmitData.appReq = &my_appl_request_struct;
                                              // This pointer is passed back to the
                                              // application-response-callback-function and
                                              // may be used there to identify the request that
                                              // caused the response callback.
SubmitData.appReqBlockSize = sizeof(my_appl_request_struct);

// Fill some custom data.
strcpy(myContextData, "this is a context data string");     // Store context data to be accessed in
                                                            // the response callback.

// customize callbackContextData
callbackContextData->custData = myContextData;         // Pass pointer to context data to
                                                       // VALUES. This pointer is passed back to the
                                                       // application in the application callback.
callbackContextData->custBlockSize = sizeof(myContextData);

// Call VCI_Submit for the first user. This actually sends the processing request
// to the Back End application.
VCI_Submit(  &reqControl1,
             &SubmitData,
             responseCallback,
             callbackContextData,
```

```
                &statusData
                );

    // Assess whether or not the call succeeded.
    ret_status_handling(&statusData);

    // In this function we poll the VMQ and call VCI_Dispatch upon event.
    poll_and_dispatch ( &reqControl1, fd );
```

4.8 Receiving Application Responses

The following code fragment shows how callbacks are implemented in the end user application to perform asynchronous processing in response to end user application requests.

Code example for an application request callback:

```
void responseCallback (  ReqCntrlT       *reqControl,
                         CallBkAppDataT   *appData,
                         AppCntxtDataT    *appCntxtData,
                         StatusDataT      *statusDataGlobal
                         )
{
    int                           responseSize;
    optEntSLegOrdrRequestT        *my_appl_req_struct;
    optEntSLegOrdrResponseT       *responseData;

    printf( "\n**************************************** \n");
    printf( "\tResponse Callback \n");
    printf( "**************************************** \n");

    switch( statusDataGlobal->techComplCode )
    {
        // This function does not handle all completion codes that may occur..
        case ELB_TECH_OK:

            // The reqControl structure contains the reqID of the request that generates
            // this callback. The reqID may be retrieved at this point to
            // determine further processing.

            // The structure appReqData of the appData parameter
            // contains the request that was sent by the application with the
            // VCI_Submit call.
            // This information can be retrieved as follows:
            my_appl_req_struct = (optEntSLegOrdrRequestT*) appData->appReqData;

            // The structure appData contains the application response data and the
            // size of response data:
            responseSize = appData->appRespBlockSize;
            responseData = (optEntSLegOrdrResponseT*) appData->appRespData;

            // The structure appCntxtData contains the application context data, which can
            // be retrieved similar to the appReqData or the appRespData.

            ////////////////////////////////////////////////////////////////////////////
            // For verification purposes send out the data from appCntxtDataT
```

```
            // and some request and response data from appData.
            ///////////////////////////////////////////////////////////////////////////
            printf("appCntxtDataT.custData = %s\n", (char*)appCntxtData->custData);
            printf("application_request_structure->basic.optCntrIdGrp.prodId = %*.*s\n",
                    PROD_ID_LEN, PROD_ID_LEN,
                    my_appl_req_struct->basic.optCntrIdGrp.prodId );
            printf("responseData->basic.ordrNo = %*.*s\n",
                    DRIV_ORDR_NO_LEN, DRIV_ORDR_NO_LEN,
                    responseData->basic.ordrNo );
            break;
        case ELB_TECH_REQ_UNSUCCESSFUL:
            // In this exception case the application must determine, based on the detailed context
            // (exception decode, application request, trading phase etc.) whether the request
            // must be resubmitted or not
            printf( "\n%s\n", statusDataGlobal->techComplText );
            // The functional completion code provides detailed information:
            printf( "%s\n", statusDataGlobal->complText );
            break;
        case ELB_TECH_PENDING_REQUEST_DELETED:
            // This exception occurs for example if an application logs out with pending
            // application requests
            printf( "\n%s\n", statusDataGlobal->techComplText );
            break;
        default:
            printf( "\n%s\n", statusDataGlobal->techComplText );
            printf( "Connection Id: %d\n", reqControl->connectionID );
            break;
    }

    // deallocation of appReqData - this assumes that the data has been malloc()ed before it was
    // passed to VCI_Submit().
    // Of course, if you used a struct that is local to a function, it must not be free()ed here -
    // however, the data
    // must be kept in scope and unchanged until the response callback has been received
    // (often, it's more convenient to malloc).
    free(appData->appReqData);
}
```

4.9 Subscribing to a Data Stream

The VCI_Subscribe call has to be used by the end user application to subscribe to data streams. Before subscribing, a session must have been established. For Xervices that implement the Broadcast Extension (see *section 2.5.2*), a valid login is required as well. The user must specify the desired data stream and a callback function.
The following list describes how the end user application can allocate and de-allocate memory when using the VCI_Subscribe and VCI_Unsubscribe:

- create subscription handler
- allocate memory for request data
- allocate memory for context data
- mark subscription handler as “pending”
- call VCI_Subscribe (asynchronous)
- if VCI_Subscribe’s techComplCode not ELB_TECH_OK *(subscribe failed)*

- de-allocate context data’s memory
- de-allocate request data’s memory
- delete subscription handler
- wait to be informed about state changes (broadcast) using the subscription callback
- depending on subscription callback’s completion code
- if ELB_TECH_OK *(broadcast)*
- no action
- if ELB_TECH_SUBSCRIBED *(subscribed)*
- mark subscription handler as “subscribed”
- if either ELB_TECH_SUBSCRIPTION_DELETED, ELB_TECH_INTERNAL_ERROR, or ELB_TECH_UNSUBSCRIBED *(unsubscribed)*
- de-allocate context data’s memory
- de-allocate request data’s memory
- delete subscription handler
- if ELB_TECH_PENDING_REQUEST_DELETED
- if subscription handler “pending” *(error while processing subscription request)*
- de-allocate context data’s memory
- de-allocate request data’s memory
- delete subscription handler
- else *(error while processing unsubscription request)*
- no action
- call VCI_Unsubscribe (asynchronous)
- no action

Code example for VCI_Subscribe

```
void subscribe_function (void)
{
    // fill the reqControl parameters
    strcpy(reqControl.appDescr, "Xetra");            // This is an optional description.
    assert(xetraDbApplId != XERVICE_YET_UNKNOWN);    // must be announced by connect callback by now
    reqControl.dbApplID = xetraDbApplId;             // Subscribe will be done for Xetra.
    strcpy(reqControl.VCIver, CVN_012);              // Set VALUES version for compatibility checking;
                                                     // VCI_VERSION is a VALUES constant.

    // At this point, the connection ID obtained as a returned parameter
    // from VCI_Connect is used to identify the session.
    reqControl.connectionID = myConnectionID;

    // For a Xervice with Broadcast Extension, it would be required to set the loginID field
    // to a valid ID that has been received via login callback, not just zero like here
```

```
    reqControl.loginId = 0;

    // Allocate memory for reqData (this memory has to be deallocated after calling VCI_Unsubscribe).
    reqData = (SubsReqDataT*) malloc( sizeof(SubsReqDataT) );

    // Fill the fields of the reqData-structure.
    // These fields should not be hardcoded, but entered by a user, or retrieved from a file.
    reqData->streamType = XETRA_PUBLIC_UNRELIABLE_MARKET_STREAM_TYPE;

    // Assign application version as defined in vld_val.h.
    reqData->applVersion     = XETR_AVN_071;
    reqData->subjectLength   = sizeof( XetraIsinSubjectT );
    reqData->subject         = &subscrSubject;

    // For subscription to Gap Notifications, this would read:
    // reqData->applVersion      = SUBJECT_GAPINFO_VERSION;
    // reqData->subjectLength    = strlen( SUBJECT_GAPINFO );
    // reqData->subject          = SUBJECT_GAPINFO;

    reqData->authorizationData           = (void*) NULL;
    reqData->authorizationDataLength     = 0;

    // The filling of the application request structure should be done at this point.
    // In general these fields are not hardcoded, but entered by a user.
    memcpy(subscrSubject, "DE0001234567", ISIN_LEN);

    statusDataGlobal = (StatusDataT*) malloc( sizeof(StatusDataT) );

    // Call VCI_Subscribe to actually subscribe to the data stream.
    VCI_Subscribe(   &reqControl,
                     reqData,
                     broadcastCallback,
                     NULL,
                     NULL,
                     statusDataGlobal
                 );

    // Assess whether or not the call succeeded
    ret_status_handling(statusDataGlobal);
}
```

4.10 Receiving Subscription Data

The following code fragment shows how callbacks are implemented in the end user application to perform asynchronous processing of received broadcasts.

Code example for a subscription response callback:

```
void broadcastCallback (      ReqCntrlT        *reqData,
                              CallBkAppDataT   *appData,
                              AppCntxtDataT    *appCntxtData,
                              StatusDataT      *statusDataGlobal
                              )
{
    int                 responseSize;
    XetraIsinSubjectT*  responseData;
```

```
int                 subsID;

// The structure "brcSubject" of the appData parameter
// contains the subject of this broadcast message
// and the subscription ID.
// This subject is identical to the subject used to subscribe to this stream,
// except that wildcards are replaced by actual values.
// A Gap Notification carries the subject SUBJECT_GAPINFO.

// The structure appData contains a pointer to the application response data and the
// size of the response data.
responseSize = appData->appReqBlockSize;
responseData = (XetraIsinSubjectT*) appData->appRespData;

// Context data may be read at this point
printf("appCntxtData.custData = %s\n", (char*)appCntxtData->custData);

switch( statusDataGlobal->techComplCode )
{
    // This function does not handle all completion codes that may occur..
    case ELB_TECH_OK:
        // at this point, the broadcast data struct may be read and processed
        // ...

        printf( "Successfully Processed\n" );
        break;
    case ELB_TECH_REQ_UNSUCCESSFUL:
        // In this exception case the application must determine, based on the detailed context
        // (exception decode, subscription request, trading phase etc.) whether the subscription
        // request must be resubmitted or not
        printf( "\n%s\n", statusDataGlobal->techComplText );
        // The functional completion code provides detailed information:
        printf( "%s\n", statusDataGlobal->complText );
        break;
    case ELB_TECH_SUBSCRIPTION_DELETED:
        // The application can try to re-issue the subscription request
    case ELB_TECH_INTERNAL_ERROR:
        // The application should disconnect and try to restart. If the error persists
        // then exception logs should be examined for more information about the problem.
    case ELB_TECH_PENDING_REQUEST_DELETED:
        // the pending subscription request did not succeed.
        break;
    case ELB_TECH_UNSUBSCRIBED:
        // This indicates a successful unsubscription
        printf( "\n%s\n", statusDataGlobal->techComplText );
        break;
    case ELB_TECH_SUBSCRIBED:
        // This indicates a successful subscription

        // store the subscription ID in the global variable
        subsID = ((SubsRespDataT*) appData->appRespData)->subsID;
        printf( "\n%s\n", statusDataGlobal->techComplText );
        break;
    default:
        printf( "\nUnknown Error! Connection Id: %d\n", reqData->connectionID );
        if (appCntxtData->custBlockSize > 0)
        {
            printf("appCntxtData.custData = %s\n", (char*)appCntxtData->custData);
        }
```

```
            printf("Application Version = %d\n", appData->applVersion);
        break;
    }
}
```

4.11 Unsubscribing from a Data Stream

The VCI_Unsubscribe entry point has to be used by the end user application to end the subscription to a data stream.

Code example for VCI_Unsubscribe:

```
void unsubscribe_function (void)
{
    // define variables in VCI-format
    ReqCntrlT       reqControl;         // request control record
    UnsubsReqDataT  reqData;            // unsubscribe request data block
    StatusDataT*    statusDataGlobal;   // VCI function completion status record

    // fill the reqControl parameters
    strcpy(reqControl.appDescr, "Xetra");     // This is an optional description.
    reqControl.dbApplID = xetraDbApplId;      // Unsubscribe will be done for Xetra.
    strcpy(reqControl.VCIver, CVN_012);       // Set VALUES version for compatibility checking;
                                              // VCI_VERSION is a VALUES constant.

    // at this point, the connection ID obtained as a returned parameter
    // from VCI_Connect is used to identify the session
    reqControl.connectionID = myConnectionID;

    // the globally stored subscription ID is needed to unsubscribe.
    reqData.subsID = subsId;

    statusDataGlobal = (StatusDataT*) malloc( sizeof(StatusDataT) );

    // call VCI_Unsubscribe
    VCI_Unsubscribe(    &reqControl,
                        &reqData,
                        NULL,
                        statusDataGlobal
                        );

    // Assess whether or not the call succeeded
    ret_status_handling(statusDataGlobal);
}
```

4.12 Logging off from an Exchange Service

The VCI_Logout entry point has to be used by the end user application to log off from an Exchange service.

Code example for VCI_Logout:

```
// Log out first user from the Back End application.
// The structure reqControl1 of type ReqCntrlT contains login context data of the first user
VCI_Logout(  &reqControl1,
             NULL,
             NULL,
             &statusData
             );

// Assess whether or not the call succeeded.
ret_status_handling(&statusData);

// In this function we poll the VMQ and call VCI_Dispatch upon event.
poll_and_dispatch ( &reqControl1, fd );

// Log out second user.
// The structure reqControl2 contains login context data of the second user
VCI_Logout(  &reqControl2,
             NULL,
             NULL,
             &statusData
             );

// Assess whether or not the call succeeded.
ret_status_handling(&statusData);

// In this function we poll the VMQ and call VCI_Dispatch upon event.
poll_and_dispatch ( &reqControl1, fd );
```

4.13 Terminating a VALUES Session

The VCI_Disconnect entry point has to be used to disconnect the end user application from VALUES.

Code example for VCI_Disconnect:

```
// allocate memory for discnctreqData (this memory has to be deallocated after disconnecting)
discnctreqData = (DiscnctReqDataT*) malloc( sizeof(DiscnctReqDataT) );

// Call VCI_Disconnect to terminate the VALUES session.
VCI_Disconnect(  &reqControl1,
                 discnctreqData,
                 &statusData
                 );

// Assess whether or not the call succeeded.
ret_status_handling(statusData);

// deallocation of connection request data
free(cnctreqData);
free(discnctreqData);
```

4.14 Auxiliary Functions of an End User Application

This section contains examples of some auxiliary functions as an end user application might contain them. They are mentioned here to make the example program complete.

```
void error_handling ( StatusDataT* statusDataGlobal )
{
    printf("Error: %s\n", statusDataGlobal->complText);
    exit(statusDataGlobal->complCode);
}


void warning_handling ( StatusDataT* statusDataGlobal )
{
    printf("Warning: %s\n", statusDataGlobal->complText);
}

void ret_status_handling ( StatusDataT* statusDataGlobal )
{
    // Assess whether or not the call succeeded.
    switch (statusDataGlobal->complSeverity)
    {
        case VCI_SUCCESS:    // VCI_SUCCESS = Successful completion.
            break;
        case VCI_FATAL:      // VCI_FATAL = Fatal error has occurred.
                             // The application should perform a shutdown.
            fatality_handling(statusDataGlobal);
            break;
        case VCI_ERROR:      // VCI_ERROR = An error has been detected.
                             // The application has to perform
                             // error-handling processing depending on which
                             // completion code has been returned.
            error_handling(statusDataGlobal);
            break;
        case VCI_WARNING:    // VCI_WARNING = A minor error occurred. The
                             // The application may have to perform error-
                             // handling depending on which completion code
                             // has been returned.
            warning_handling(statusDataGlobal);
            break;
    }   // end switch
    return;
}
```

5 VALUES API Completion Codes (Call Interface)

The VALUES API Call Interface communicates status information with a completion code data field. Each entry point generates a number of different completion codes. Some codes are shared among the entry points.

For a list of VALUES API functional completion codes please refer to the Exchange specific volumes.

Table 5.1 lists for all valid VALUES API technical completion codes the completion text and a description. To access these completion codes and text decodes, the constant definitions in the completion code header file should be used. Please refer to *section 7* for a list of header files delivered with the Exchange Service software.

Completion Text (techComplText)	Description
SUCCESSFUL COMPLETION	Indicates successful completion status of an entry point.
MAXIMUM NUMBER OF CONNECTIONS REACHED	The maximum number of VALUES sessions on a workstation or MISS is reached.
USER IS NOT ALLOWED TO USE THE EXCHANGE SERVICE OR USER IS NOT REGISTERED	The user ID specified is not valid.
INVALID PASSWORD	The password specified is not valid.
A PARAMETER WAS EITHER INVALID OR HAD INVALID CHARACTERS	One or more specified parameters are invalid. Please check the exception log file for details.
INVALID USER OR PASSWORD	One or more specified parameters are invalid. Please check the exception log file for details.
INVALID PARAMETER PASSED	One or more specified parameters are invalid. Please check the exception log file for details.
INTERNAL ERROR OCCURRED	A VALUES internal error occurred, please check the exception log file for details and contact the help desk in case the problem cannot be solved.
APPLICATION ALREADY CONNECTED	A session is already established.
REQUEST NOT SUCCESSFULLY PROCESSED	An exception has occurred while processing the request. Please check the Exchange application completion code for details.
APPLICATION NOT CONNECTED	A session must be established first.
PENDING REQUEST DELETED	The application disconnected or logged out while requests where pending. Alternatively, a non-transparent failover might have occurred, requests must be re-transmitted.

Table 5.1 - VALUES API Completion Codes

Completion Text (techComplText)	Description
SUBSCRIPTION DELETED	The application disconnected while subscriptions where active. For a Xervice that implements Broadcast Extensions, this error can also occur in case of losing the login context that the subscription refers to.
MAXIMUM NUMBER OF LOGINS EXCEEDED	The maximum number of logins per Exchange Service is exceeded.
EXCHANGE SERVICE NOT AVAILABLE	An Exchange Service is not available. Please contact the operator.
TOO MANY PENDING REQUESTS IN QUEUE	The limit of pending requests is reached. Wait for responses before submitting more requests.
USER NOT LOGGED IN	User not yet authorized.
TECHNICAL SERVICES NOT AVAILABLE	GATE (Technical Services) is unavailable. Please contact the operator.
USER LOGGED IN SUCCESSFULLY	User has been authorized successfully.
USER LOGGED OUT SUCCESSFULLY	User logged out from Exchange Service successfully.
STREAM SUBSCRIBED	The specified subscription is activated.
STREAM UNSUBSCRIBED	The specified subscription is de-activated.
UNMAPPABLE MESSAGE DISCARDED	A response was received with no corresponding pending request. This event may occur during recovery of workstation or MISS (failover).
NONTRANSPARENT FAILOVER	A non-transparent failover to the secondary MISS has occurred. Requests may have been lost, recovery through inquiry may be required.
VALUES CALLS ARE NOT REENTRANT	A VALUES entry point has been used while another VALUES call in the same process was still active.
EXCHANGE SERVICE AVAILABLE	An Exchange Service has become available.
STREAM NOT SUBSCRIBED	The subscription is not active.

Table 5.1 - VALUES API Completion Codes

6 VALUES API Call Interface Field Descriptions

This is a description of the data fields passed to or retrieved from the VALUES API Call Interface. The first section gives an overview and provides general information. The second section lists all data fields of the Call Interface.

6.1 Overview

The VALUES API field description lists all data fields by name and provides a detailed field description. Field characteristics are detailed and a guideline on how to initialize data fields is given. Valid values for the individual field as well as a textual description of field level rules are given where applicable.

6.1.1 Field Characteristics

The characteristics of each field are detailed giving the following information.

Type — Denotes the data type of the field. Valid data types are the ANSI C types listed in *Table 6.1*.

Value/Reference — Defines the passing mechanism used for this field, passed by value (Value) or passed by reference (Reference).

Type	Description	Valid Values / Range
int	An integer, typically reflecting the natural size of integers on the host machine.	platform dependent (p/d)
char [ ]	Text; in C this is a reference to text stored in an array. ‘Format’ indicates the size of this array.	alphanumeric characters, no special characters allowed
char	Single byte, capable of holding one character in the local character set.	platform dependent (p/d)
void*	Reference to a data block of unspecified structure.	platform dependent (p/d)

Table 6.1 - VALUES API Data Types

Format — Defines the format and size of the data field. The format for types that have platform dependent sizes is marked as "p/d".

Format constants are defined in data definition header files published with the VALUES API software. Please refer to *section 7* for a list of VALUES API data definitions header files.

Note: Character values used in application requests are not '\0' terminated.

6.1.2 Initialization Guideline

It is recommended to initialize all unused or optional data fields before passing them to the VALUES API. *Table 6.2* gives a guideline on how to initialize the different data types. Unused character fields in application requests should be explicitly set to "spaces".

Type	Initialization
int	0
char [ ]	array filled with spaces
char	SPACE
void*	NULL

Table 6.2 - Data Field Initialization Guideline

6.1.3 Template for the Call Interface Field Descriptions

Name of field

Description	Description of the Call Interface Field		
Where Used	Used Entry points where this field is used		
Characteristics	Type	Value/Reference	Format
Valid Values			

6.2 Call Interface Field Descriptions

6.2.1 appDescr (ReqCntrlT)

Description	This field is reserved for future use.		
Where Used	n/a		
Characteristics	Type	Value/Reference	Format
	char [ ]	Value	MAX_APPDSCR

Valid Values — Not Relevant

6.2.2 applClass (XerviceInfoT)

Description — This field specifies the Xervice Class assigned to a Xervice. Please refer to *section 2.9* for a description of the Xervice Class concept.

Where Used — In application callbacks

Characteristics

Type	Value/Reference	Format
int	Value	p/d

Valid Values — Defined in exid.h, please refer to *section 7* for information on data definitions and header files.

6.2.3 applPrevVersion (CallBkAppDataT)

Description — This field supplies an AVN that is supported by an Exchange Application in terms of backwards compatibility. If no backwards compatibiliy is available, this field contains the same value as applVersion.

Where Used — In application callbacks.

Characteristics

Type	Value/Reference	Format
int	Value	p/d

Valid Values — Please refer to the description of this field in the Exchange specific volumes.

6.2.4 applVersion (LoginReqDataT, SubsReqDataT, CallBkAppDataT, XerviceInfoT)

Description — This field identifies the version of the application request (AVN).

Where Used	VCI_Subscribe VCI_Login In application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Please refer to the description of this field in the Exchange specific volumes.		

6.2.5 appReq (SubmitReqDataT)

Description	This is a pointer to the application request data structure passed to VCI_Submit.		
Where Used	VCI_Submit		
Characteristics	Type	Value/Reference	Format
	void*	Reference	p/d
Valid Values	Application data according to specific Exchange application.		

6.2.6 appReqBlockSize (CallBkAppDataT, SubmitReqDataT)

Description	This field specifies the number of bytes of the application request data block passed to VCI_Submit by the end user application This field is returned from VALUES in application callbacks.		
Where Used	VCI_Submit In application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Use actual size of application request structure.		

6.2.7 appReqData (CallBkAppDataT)

Description: This data structure is returned from VALUES in application callbacks at receipt of responses. It contains the application request corresponding to the response.

Where Used: In application callbacks

Characteristics:

Type	Value/Reference	Format
void*	Reference	p/d

Valid Values: Application data according to specific Exchange application.

6.2.8 appRespBlockSize (CallBkAppDataT)

Description: This field specifies the number of bytes of the application response data block populated by VALUES and passed to the end user application through application callbacks.

Where Used: In application callbacks

Characteristics:

Type	Value/Reference	Format
int	Value	p/d

Valid Values: Use actual size of application response structure.

6.2.9 appRespData (CallBkAppDataT)

Description: This is a pointer to the application response data structure populated by VALUES and returned to the end user application through application callbacks.

Where Used: In application callbacks

Characteristics	Type	Value/Reference	Format
	void*	Reference	p/d

Valid Values — Application data according to specific Exchange application.

6.2.10 authorizationData (LoginReqDataT, SubsReqDataT)

Description — This is a pointer to the Exchange application authorization data and used for passwords and additional Exchange application specific data.

Where Used — VCI_Login

VCI_Subscribe (for future use)

Characteristics	Type	Value/Reference	Format
	void*	Reference	p/d

Valid Values — Application authorization data according to specific Exchange application.

6.2.11 authorizationDataLength (LoginReqDataT, SubsReqDataT)

Description — This field specifies the length of the field authorizationData.

Where Used — VCI_Login

VCI_Subscribe (for future use)

Characteristics	Type	Value/Reference	Format
	int	Value	p/d

Valid Values — Size of the field authorizationData.

6.2.12 brcSubject (CallBkAppDataT)

Description: This field defines the subject data structure. The subject data structure is populated by the end user application and passed to VALUES as a byte block. Subject structures are defined with each subscription application request in the Exchange application specific volumes.

Where Used: VCI_Subscribe

In application callbacks

Characteristics	Type	Value/Reference	Format
	void*	Reference	n/a

Valid Values: For definition of subject structures please refer to the Exchange application specific volumes. Subject structures are defined in the description of each subscription application request.

6.2.13 closure (LoginReqDataT)

Description: This field is reserved for future use.

Where Used: n/a

Characteristics	Type	Value/Reference	Format
	char	Value	1

Valid Values: Not Relevant

6.2.14 complCode (statusDataT)

Description: This field contains a code that describes the completion status of an application request. This completion status is generated by the Exchange service and passed to the application callback by VALUES.

Where Used: All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	int	Value	p/d

Valid Values: Value varies according to the given application request. For a list of application request completion codes and corresponding decodes please refer to the Exchange application specific volumes.

6.2.15 complSeverity (statusDataT)

Description: This field defines the severity of the completion code (complCode). Value varies according to the circumstances in which the processing occurred.

Where Used: All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	int	Value	p/d

Valid Values:

VCI_SUCCESS

VCI_WARNING

VCI_ERROR

VCI_FATAL

6.2.16 complText (statusDataT)

Description: This field is the decode of a specific completion code. This is a textual description of the completion code

Where Used: All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	char [ ]	Value	ELB_MAX_STRING

Valid Values: Value varies according to the given application request. For a list of application request codes and decodes please refer to the Exchange application specific volumes.

6.2.17 connectionID (ReqCntrlT, CnctRespDataT)

Description

This field identifies a session which is unique per application using VALUES. The field is populated by VALUES in the response message of the VCI_Connect entry point. For all other entry points the field must be populated by the application in the request control block with the connection ID obtained through VCI_Connect.

Where Used

All Call Interface entry points

Characteristics

Type	Value/Reference	Format
int	Value	p/d

Valid Values

Valid connection ID obtained through VCI_Connect.

6.2.18 custBlockSize (AppCntxtDataT)

Description

This field specifies the number of bytes of the custom data block (custData field) passed to VALUES by the end user application and returned through the application callback.

Where Used

VCI_Connect

VCI_Login

VCI_Submit

VCI_Subscribe

In application callbacks

Characteristics

Type	Value/Reference	Format
int	Value	p/d

Valid Values

Numeric

6.2.19 custData (AppCntxtDataT)

Description	This field specifies a pointer to the custom data block passed to VALUES by the end user application and returned through the application callback.		
Where Used	VCI_Connect VCI_Login VCI_Submit VCI_Subscribe In application callbacks		
Characteristics	Type	Value/Reference	Format
	void*	Reference	p/d
Valid Values	User specific context data.		

6.2.20 dbApplID (ReqCntrlT)

Description	This field uniquely identifies Exchange Services to login or logout from, or to submit/subscribe to. This field is filled by the Connection callback and the end user application.		
Where Used	VCI_Login VCI_Logout VCI_Submit VCI_Subscribe in application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Any integer received by the application callback for Connect events (see *section 3.3.3* for details).		

6.2.21 exchApplId (XerviceInfoT)

Description	This field contains the Market Identification Code (MIC) that uniquely identifies a certain Exchange, e.g. “XEUR” for Eurex or “XVIE” for Xetra Vienna.		
Where Used	in application callbacks		
Characteristics	Type	Value/Reference	Format
	char[ ]	Value	MAX_APPLID
Valid Values	Valid MIC codes. Please note that this is not a '\0' terminated field.		

6.2.22 exchDscrName (XerviceInfoT)

Description	This field contains a readable free text identifier for an Exchange. It can be used to display a menu to the end user.		
Where Used	in application callbacks		
Characteristics	Type	Value/Reference	Format
	char[ ]	Reference	MAX_DSCRNAME
Valid Values	Free text. This field contains a '\0' terminated C string.		

6.2.23 funcResult (LoginRespDataT)

Description	This field is reserved for future use.		
Where Used	n/a		
Characteristics	Type	Value/Reference	Format
	int	Value	n/a
Valid Values	Not Relevant		

6.2.24 loginID (ReqCntrlT, LoginRespDataT)

Description	This field contains an unique identifier for each successful login. The loginID is unique per Exchange Service. The field is populated by VALUES and returned to the end user application with the response of a successful login request. The loginID obtained must be passed to VALUES by the end user application for each subsequent call to VCI_Submit or VCI_Logout. This also appplies to VCI_Subscribe if Broadcast Extension is implemented by the Xervice. This field is used to authenticate a specific user.		
Where Used	VCI_Login VCI_Logout VCI_Submit		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Valid loginID obtained by VCI_Login		

6.2.25 password (CnctReqDataT)

Description	This field is used to authorize connection to VALUES. The password is specified by the end user application and passed to VALUES. The characters that can be entered into this field are all characters allowed for system passwords of the platform that the MISS is running on.		
Where Used	VCI_Connect		
Characteristics	Type	Value/Reference	Format
	char [ ] (see above for special characters)	Value	MAX_PWDID
Valid Values	A valid password registered on the MISS.		

6.2.26 prodMode (CnctRespDataT)

Description	This field specifies the production mode of Exchange applications: production, simulation, or development. This field is returned to the end user application through the response of VCI_Connect.		
Where Used	VCI_Connect		
Characteristics	Type	Value/Reference	Format
	char	Value	1
Valid Values	VCI_AREA_PROD		
	VCI_AREA_SIM		
	VCI_AREA_DEV		

6.2.27 reqID (ReqCntrlT)

Description	This field specifies the application request to be submitted through VCI_Submit within a context of an Exchange Service. Each application request is assigned a unique request ID. The end user application specifies the request ID.		
Where Used	VCI_Submit		
	In application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	For a complete list of available request Ids, please refer to the Exchange application specific volumes.		

6.2.28 resubmitFlag (ReqCntrlT)

Description	This flag is used to indicate whether a resubmitted application request was processed or not. If processing did occur, then this field is set to RESUBMISSION_PROCESSED and data is available in the application response. If processing did not occur - either because of an error or because processing was finished in response to a different instance of the associated request, then this field is set to RESUBMISSION_NOT_PROCESSED and no data is available in the application response.		
Where Used	In application callbacks. Refer to Exchange Application Software specific Volume.		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	RESUBMISSION_PROCESSED RESUBMISSION_NOT_PROCESSED		

6.2.29 resubmitNo (ReqCntrlT)

Description	This field uniquely identifies the corresponding resubmitted application request.		
Where Used	VCI_Submit In application callbacks. Refer to Exchange Application Software specific Volume.		
Characteristics	Type	Value/Reference	Format
	unsigned int	Value	p/d
Valid Values	Numeric		

6.2.30 streamType (CallbkAppDataT, SubsReqDataT)

Description	This field specifies the broadcast data stream that the application wants to subscribe to. Each data stream defines its own streamType which is unique within the context of an Exchange application. This field is populated by the end user application and passed to VALUES.		
Where Used	VCI_Subscribe In application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	For valid stream types please refer to the data definition files published with the VALUES API software. See *section 7* for information on data definition files. Also, please refer to the Exchange application specific volumes for further information on subscriptions and broadcast stream types.		

6.2.31 subject (CallBkAppDataT)

Description	This field is reserved for future use.		
Where Used	n/a		
Characteristics	Type	Value/Reference	Format
	BcastSubjectT*	n/a	n/a
Valid Values	Not Relevant		

6.2.32 subject (SubsRegDataT)

Description	This field defines the subject data structure. The subject data structure is populated by the end user application and passed to VALUES as a byte block. Subject structures are defined with each subscription application request in the Exchange application specific volumes.

Where Used	VCI_Subscribe		
Characteristics	Type	Value/Reference	Format
	void*	Reference	p/d
Valid Values	For definition of subject structures please refer to the Exchange application specific volumes. Subject structures are defined in the description of each subscription application request.		

6.2.33 subjectLength (SubsReqDataT)

Description	This field defines the byte length of the broadcast subject. This field is filled by the end user application and passed to VALUES.		
Where Used	VCI_Subscribe		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Use actual size of subject data structure.		

6.2.34 subsID (CallBkAppDataT, SubsRespDataT)

Description	This field uniquely identifies a subscription. The field is populated by VALUES in the response of a successful subscription request. The end user application specifies the subsID to unsubscribe from a specific data stream.		
Where Used	VCI_Unsubscribe In application callbacks		
Characteristics	Type	Value/Reference	Format
	int	Value	p/d
Valid Values	Valid subsID obtained by VCI_Subscribe		

6.2.35 subsSubject (SubsRegDataT)

Description: This field is reserved for future use.

Where Used: n/a

Characteristics	Type	Value/Reference	Format
	BcastSubjectT*	n/a	n/a

Valid Values: Not Relevant

6.2.36 techComplCode (StatusDataT)

Description: This field contains a code that describes the completion status of either the Call Interface call or GATE. This field is filled by VALUES on return from a call to the interface.

Where Used: All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	int	Value	p/d

Valid Values: Value varies according to Call Interface entry point. Please refer to *section 5* for a list of completion codes.

6.2.37 techComplSeverity (StatusDataT)

Description: This field defines the severity of the completion code (techComplCode). Value varies according to the circumstances in which the processing occurred; e.g., successful completion of processing results in VCI_SUCCESS severity.

Where Used: All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	int	Value	p/d

Valid Values
VCI_SUCCESS
VCI_WARNING
VCI_ERROR
VCI_FATAL

6.2.38 techComplText (StatusDataT)

Description
This field is the decode of a specific completion code. This is a textual description of the completion code.

Where Used
All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	char [ ]	Value	ELB_MAX_STRING

Valid Values
Value varies according to Call Interface entry point. Please refer to *section 5* for a list of completion codes and decodes.

6.2.39 userID (CnctReqDataT)

Description
This field contains the user identification to authenticate access to VALUES. The field is populated by the end user application and passed to VALUES.

Where Used
VCI_Connect

Characteristics	Type	Value/Reference	Format
	char[ ]	Value	MAX_USRID

Valid Values
A valid user ID known on the MISS.

6.2.40 userID (LoginReqDataT)

Description	This field contains the user identification to authenticate access to an Exchange Service. The field is populated by the end user application and passed to VALUES.		
Where Used	VCI_Login		
Characteristics	Type	Value/Reference	Format
	char [ ]	Value	LOGIN_MAX_ USRID
Valid Values	A valid user ID known to the Exchange Service.		

6.2.41 userID (SubsReqDataT)

Description	This field contains the user identification to authenticate access to broadcast streams of an Exchange application. The field is populated by the end user application and passed to VALUES.		
Where Used	VCI_Subscribe		
Characteristics	Type	Value/Reference	Format
	char[ ]	Value	SUBS_MAX_USERID
Valid Values	A valid user ID known to the Exchange application (for future use).		

6.2.42 VCIver (ReqCntrlT)

Description	This field contains the version number (CVN) of the VALUES API Call Interface. The CVN is used to support Call Interface backwards compatibility.
Where Used	All Call Interface entry points

Characteristics	Type	Value/Reference	Format
	char [ ]	Value	MAX_VCIVER

Valid Values CVN_012

6.2.43 VMQname (CnctRespDataT)

Description: This field identifies the VALUES message queue. The field is populated by VALUES through the VCI_Connect entry point. The end user application uses VMQname to address the VALUES message queue; i.e., to check for pending events.

Where Used: VCI_Connect

Characteristics	Type	Value/Reference	Format
	char[ ]	Value	MAX_VMQNAM

Valid Values: Left-aligned and '\0' terminated queue name.

7 Data Definitions

This section provides a list of the VALUES header files needed to use the VALUES API Call Interface (technical components). These header files contain the constants, enumeration types and structure definitions necessary to use the VALUES API Call Interface. *Table 7.1* gives the description of each header file. All header files are provided in electronic format with the Exchange application software.

The header files necessary to use the VALUES API application requests are listed in the Exchange application specific volumes.

File	Description
Values.h	Contains Call Interface function prototypes, constant definitions and other structure definitions needed to use the VALUES API Call Interface.
elbcodetech.h	This header file is a collection of all technical exception codes that may be returned by the VALUES API. Constant definitions are provided for each exception code. The decodes of the technical exception codes are provided with the file msg.dat which is included in the VALUES API software distribution.
subject.h	Contains constant definitions for subscriptions (e.g., subject wildcard, GAPINFO special subject).
exid.h	Contains definitions of preprocessor constants for Xervice Classes. The preprocessor constants for individual Xervices, which have become obsolete with the introduction of the Xervice Class concept, will be removed from this file in a later release. They should not be used any longer.

Table 7.1 - VALUES API Header Files

8 Glossary

Term	Explanation
API	Application Programming Interface.
Application callback	End user application functions invoked by VALUES on receipt of an application response or other events. The end user application defines which callback is to be invoked by “registering” the callback.
Application dispatch loop	The application dispatch loop is implemented in the end user application (usually the main processing loop). It receives notification of an event on the VMQ when Exchange applications write VALUES events such as response events, subscription events, notifications and exception events to the VMQ. Upon event notification from the VMQ, the application dispatch loop will trigger VCI_Dispatch to retrieve the event from the VMQ.
Application request	Application requests are functional entry points to Exchange application services. Application requests are always used in conjunction with the VALUES API Call Interface to provide the functional information required to access Exchange application services.
Application response	Application responses are received by the end user application as a dedicated single response to a specific request. The application response contains the results of Exchange Service request processing and is delivered asynchronously to the end user application.
Asynchronous processing	An asynchronous entry point does not block until its processing is completed. Asynchronous entry points return to the end user application immediately with a status of the processing request. Processing responses are received at a later point in time (i.e., asynchronous to the processing request).
AVN	Application Version Number. Each application request, subscription request, application response and broadcast contains an application version number which defines the version of the associated Exchange application.
Back End	General term for any system based at Deutsche Börse providing centralized services to members having access to it, e.g. Xetra®.
BESS	Back End related Subsystem. The exchange specific subsystem of the MISS.
Broadcast	A broadcast is information disseminated to the members by an Exchange application (e.g., Trade Confirmation, Order Confirmation). Broadcasts are sent across different data streams, each data stream contains a specific type of information.

Table 8.1 - Glossary

Term	Explanation
Broadcast gap stream	Broadcast gaps can occur on different types of broadcast streams. The broadcast gap stream can be used to detect missing broadcasts and to inform applications of the potential loss of a broadcast. Using the Gap Info subject (refer to *section 2.7* for details) should be preferred wherever possible.
Call Interface	The Call Interface contains the technical entry points to Exchange application services. The Call Interface is a set of system subroutines called by an application to access Exchange application services.
Completion code	The VALUES API Call Interface communicates status information with a completion code data field. Each entry point may generate a number of different completion codes.
CCP	Central Counterparty
CVN	Call Interface Version Number. Each Call Interface entry point contains a Call Interface version number.
Data streams	Data streams are the vehicle used to disseminate information of a specific type to members. Each type of information defines its own data stream.
DBAG	Deutsche Börse AG
End user application	End user application is any application which receives services via the VALUES API.
Entry point	The VALUES API Call Interface consists of a fixed number of technical entry points, which are used to establish a session, transmit application requests, request data and receive responses.
Eurex	European Exchange. Electronic trading and clearing system for the derivatives market (options and futures).
Eurex US	Eurex's US Exchange
Exception event	An event sent to the VALUES API in case of exceptions with the connection (e.g., network exceptions, Exchange application exception, etc.).
Exchange application	Application provided by an Exchange that provides one or more services to an end user application.
Exchange Service	See Exchange application.
Field usage	Field usage specifies field information such as whether a field is mandatory, optional or occurs multiple times. Field usages are specified in application request descriptions.
Front End	Any member's local system providing access to an Exchange application.

Table 8.1 - Glossary

Term	**Explanation**
GATE	GATE (Generic Access to Exchanges) are processes running on members' workstation and MISS to provide the technical infrastructure for VALUES and to communicate with the Exchange application.
MAA	Member Assembled Application. It is a third party developed end user application which runs on VALUES API.
Member	Market participant.
Member applications	See MAA.
MISS	Member Integration System Server
Paging	The paging mechanism in the VALUES API allows handling of large response data sets while limiting the size of messages transferred.
Request ID	The request ID is used to identify a specific application request to Exchange Services.
Request Management Service	Provides access to Exchange Service. A request consists of a single application request and a single application response. Used to trigger processing by an Exchange Service.
Response event	Answers to end user application requests sent to Exchange Services.
Session	A VALUES session is a control technique for managing communications between an end user application and VALUES using GATE. The communication between an end user application and an Exchange application is based on a session. Only one VALUES session can be established per application process.
Subscription	Mechanism used to request event-driven broadcast information from Exchange applications. Subscriptions can be started and stopped. Broadcast information requested through subscription arrives asynchronously. Subscription requests are sent to Exchange applications specifying the desired data stream. Subsequently all new broadcast data on the specified stream is sent to the end user application asynchronously.
Synchronous processing	Synchronous Call Interface entry points block until VALUES internal processing completes; i.e., associated processing request completes with receipt of a processing response.
Technical Services	see "GATE".
User	A user for the interface specification is defined as any member, trader, or participant who receives Exchange services via the VALUES API.

Table 8.1 - Glossary

Term	Explanation
User interface Event Queue	Event queue of any event drives end user application. For example, a GUI application uses an event queue to handle mouse or keyboard input events.
VALUES	Virtual Access Link Using Exchange Services
VALUES API	The VALUES API is a set of system subroutines and data structures used to communicate with Exchange Services.
VALUES events	A VALUES event is an indication of application responses, broadcast information, notifications or exceptions received by the VALUES Message Queue (VMQ).
VCI	VALUES Call Interface. See “Call Interface”.
VMQ	VALUES Message Queue. It temporarily stores communication channel, application requests, responses and broadcasts which are transmitted between VALUES and GATE.
WS	Workstation.
Xervice	A Xervice is a service offered to a VALUES-based Front End application by a certain electronic exchange. A single exchange can offer multiple Xervices. See *section 2.9* for details.
Xervice Class	A Xervice Class is an identifier that is equal across Xervices which offer identical VALUES functionality, given the Application Version Number is also equal. See *section 2.9* for details.
Xetra®	Exchange Electronic Trading. Deutsche Börse’s electronic trading system for cash markets.
Xontro	BrainTrade’s Floor Trading System

Table 8.1 - Glossary