Avaya Aura® Contact Center / Avaya Contact Center Select
Host Data Exchange Programmer's Guide

Release 7.1.1

Issue 0.1

October 2020

Notice

While reasonable efforts have been made to ensure that the information in this document is complete and accurate at the time of printing, Avaya assumes no liability for any errors. Avaya reserves the right to make changes and corrections to the information in this document without the obligation to notify any person or organization of such changes.

Documentation disclaimer

“Documentation” means information published by Avaya in varying mediums which may include product information, operating instructions and performance specifications that Avaya may generally make available to users of its products and Hosted Services. Documentation does not include marketing materials. Avaya shall not be responsible for any modifications, additions, or deletions to the original Published version of documentation unless such modifications, additions, or deletions were performed by Avaya. End User agrees to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation, to the extent made by End User.

Link disclaimer

Avaya is not responsible for the contents or reliability of any linked websites referenced within this site or documentation provided by Avaya. Avaya is not responsible for the accuracy of any information, statement or content provided on these sites and does not necessarily endorse the products, services, or information described or offered within them. Avaya does not guarantee that these links will work all the time and has no control over the availability of the linked pages.

Warranty

Avaya provides a limited warranty on Avaya hardware and software. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya’s standard warranty language, as well as information regarding support for this product while under warranty is available to Avaya customers and other parties through the Avaya Support website: http://support.avaya.com or such successor site as designated by Avaya. Please note that if you acquired the product(s) from an authorized Avaya Channel

Partner outside of the United States and Canada, the warranty is provided to you by said Avaya Channel Partner and not by Avaya.

Licenses

THE SOFTWARE LICENSE TERMS AVAILABLE ON THE AVAYAWEBSITE, HTTP://SUPPORT.AVAYA.COM/LICENSEINFO

OR SUCH SUCCESSOR SITE AS DESIGNATED BY AVAYA, ARE APPLICABLE TO ANYONE WHO DOWNLOADS, USES AND/OR INSTALLS AVAYA SOFTWARE, PURCHASED FROM AVAYA INC., ANY AVAYA AFFILIATE, OR AN AVAYA CHANNEL PARTNER (AS APPLICABLE) UNDER A COMMERCIAL AGREEMENT WITH AVAYA OR AN AVAYA CHANNEL PARTNER. UNLESS OTHERWISE AGREED TO BY AVAYA IN WRITING, AVAYA DOES NOT EXTEND THIS LICENSE IF THE SOFTWARE WAS OBTAINED FROM ANYONE OTHER THAN AVAYA, AN AVAYA AFFILIATE OR AN AVAYA CHANNEL PARTNER; AVAYA RESERVES THE RIGHT TO TAKE LEGAL ACTION AGAINST YOU AND ANYONE ELSE USING OR SELLING THE SOFTWARE WITHOUT A LICENSE. BY INSTALLING, DOWNLOADING OR USING THE SOFTWARE, OR AUTHORIZING OTHERS TO DO SO, YOU, ON BEHALF OF YOURSELF AND THE ENTITY FOR WHOM YOU ARE INSTALLING, DOWNLOADING OR USING THE SOFTWARE (HEREINAFTER REFERRED TO INTERCHANGEABLY AS “YOU” AND “END USER”), AGREE TO THESE TERMS AND CONDITIONS AND CREATE A BINDING CONTRACT BETWEEN YOU AND AVAYA INC. OR THE APPLICABLE

AVAYA AFFILIATE (“AVAYA”).

Avaya grants you a license within the scope of the license types described below, with the exception of Heritage Nortel Software, for which the scope of the license is detailed below. Where the order documentation does not expressly identify a license type, the applicable license will be a Designated System License. The applicable number of licenses and units of capacity for which the license is granted will be one (1), unless a different number of licenses or units of capacity is specified in the documentation or other materials available to you. “Designated Processor” means a single stand-alone computing device. “Server” means a Designated Processor that hosts a software application to be accessed by multiple users.

License type(s)

Named User License (NU). You may: (i) install and use the Software on a single Designated Processor or Server per authorized Named User (defined below); or (ii) install and use the Software on a Server so long as only authorized Named Users access and use the Software. “Named User”, means a user or device that has been expressly authorized by Avaya to access and use the Software. At Avaya’s sole discretion, a “Named User” may be, without limitation, designated by name, corporate function (e.g., webmaster or helpdesk), an e-mail or voice mail account in the name of a person or corporate function, or a directory entry in the administrative database utilized by the Software that permits one user to interface with the Software.

Copyright

Except where expressly stated otherwise, no use should be made of materials on this site, the Documentation, Software, Hosted Service, or hardware provided by Avaya. All content on this site, the documentation, Hosted Service, and the Product provided by Avaya including the selection, arrangement and design of the content is owned either by Avaya or its licensors and is protected by copyright and other intellectual property laws including the sui generis rights relating to the protection of databases. You may

not modify, copy, reproduce, republish, upload, post, transmit or distribute in any way any content, in whole or in part, including any code and software unless expressly authorized by Avaya. Unauthorized reproduction, transmission, dissemination, storage, and or use without the express written consent of Avaya can be a criminal, as well as a civil offense under the applicable law.

Third Party Components

“Third Party Components” mean certain software programs or portions thereof included in the Software or Hosted Service may contain software (including open source software) distributed under third party agreements (“Third Party Components”), which contain terms regarding the rights to use certain portions of the Software (“Third Party Terms”). As required, information regarding distributed Linux OS source code (for those Products that have distributed Linux OS source code) and identifying the copyright holders of the Third Party Components and the Third Party Terms that apply is available in the Documentation or on Avaya’s website at:

http://support.avaya.com/Copyright or such successor site as designated by Avaya. You agree to the Third Party Terms for any such Third Party Components.

THIS PRODUCT IS LICENSED UNDER THE AVC PATENT PORTFOLIO LICENSE FOR THE PERSONAL USE OF A CONSUMER OR OTHER USES IN WHICH IT DOES NOT RECEIVE REMUNERATION TO (i) ENCODE VIDEO IN COMPLIANCE WITH THE AVC STANDARD ("AVC VIDEO") AND/OR (ii) DECODE AVC VIDEO THAT WAS ENCODED BY A CONSUMER ENGAGED IN A PERSONAL ACTIVITY AND/OR WAS OBTAINED FROM A VIDEO PROVIDER LICENSED TO PROVIDE AVC VIDEO. NO LICENSE IS GRANTED OR SHALL BE IMPLIED FOR ANY OTHER USE. ADDITIONAL INFORMATION MAY BE OBTAINED FROM MPEG LA, L.L.C. SEE HTTP://WWW.MPEGLA.COM.

Note to Service Provider

The Product or Hosted Service may use Third Party Components subject to Third Party Terms that do not allow hosting and require a Service Provider to be independently licensed for such purpose. It is your responsibility to obtain such licensing.

Preventing Toll Fraud

"Toll Fraud" is the unauthorized use of your telecommunications system by an unauthorized party (for example, a person who is not a corporate employee, agent, subcontractor, or is not working on your company's behalf). Be aware that there can be a risk of Toll Fraud associated with your system and that, if Toll Fraud occurs, it can result in substantial additional charges for your telecommunications services.

Avaya Toll Fraud intervention

If you suspect that you are being victimized by Toll Fraud and you need technical assistance or support, call Technical Service Center Toll Fraud Intervention Hotline at +1-800-643-2353 for the United States and Canada. For additional support telephone numbers, see the Avaya Support website: http://support.avaya.com or such successor site as designated by Avaya. Suspected security vulnerabilities with Avaya products should be reported to Avaya by sending mail to: securityalerts@avaya.com.

Trademarks

The trademarks, logos and service marks ("Marks") displayed in this site, the Documentation, Hosted Service(s), and Product(s) provided by Avaya are the registered or unregistered Marks of Avaya, its affiliates, or other third parties. Users are not permitted to use such Marks without prior written consent from Avaya or such third party which may own the Mark. Nothing contained in this site, the Documentation, Hosted Service(s) and Product(s) should be construed as granting, by implication, estoppel, or otherwise, any license or right in and to the Marks without the express written permission of Avaya or the applicable third party.

Avaya is a registered trademark of Avaya Inc.

All non-Avaya trademarks are the property of their respective owners. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

All non-Avaya trademarks are the property of their respective owners, and "Linux" is a registered trademark of Linus Torvalds.

Downloading Documentation

For the most current versions of Documentation, see the Avaya Support website: http://support.avaya.com or such successor site as designated by Avaya.

Contact Avaya Support

See the Avaya Support website: http://support.avaya.com for Product or Hosted Service notices and articles, or to report a problem with your Avaya Product or Hosted Service. For a list of support telephone numbers and contact addresses, go to the Avaya Support website: http://support.avaya.com (or such successor site as designated by Avaya), scroll to the bottom of the page, and select Contact Avaya Support.

Contents

Chapter 1: Introduction

Purpose

This document provides information for developing applications that use the Host Data Exchange (HDX) Application Programming Interface (API). The HDX API provides both a Common Object Request Broker Architecture (CORBA) and a C-API type programming interface.

The document provides information on the HDX Software Development Kit (SDK). The HDX SDK distributes the development and deployments environment.

Intended audience

This document is intended for people who want to develop applications that use the HDX API.

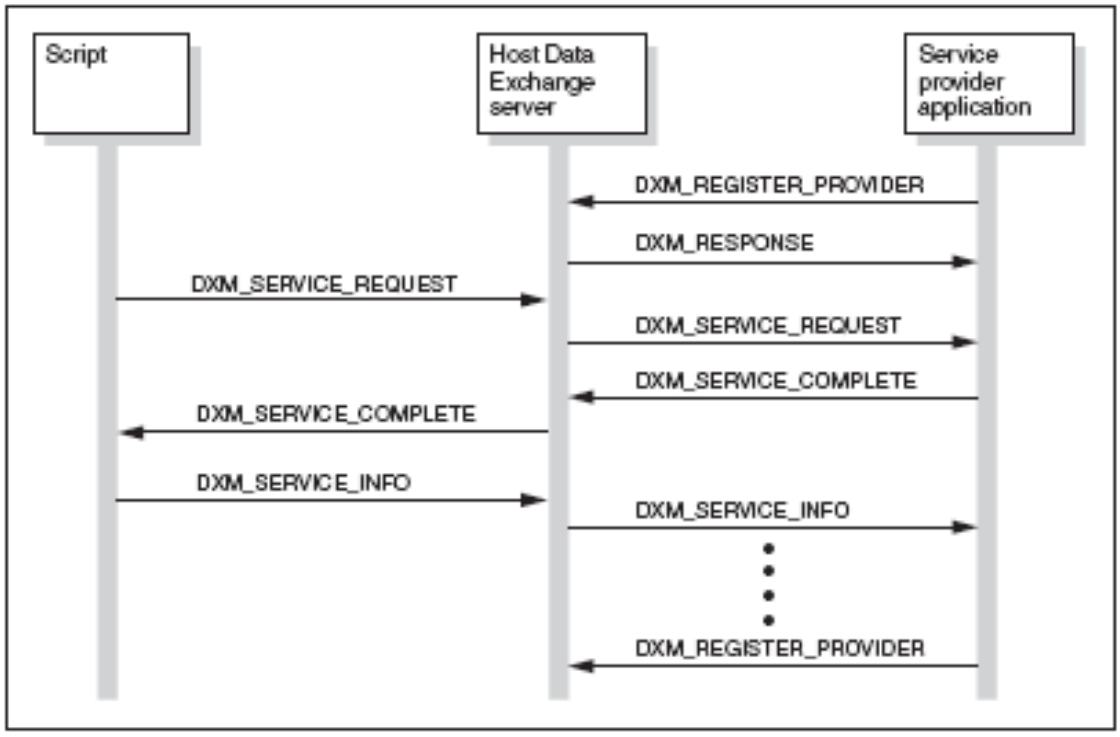
Support

Visit the Avaya Support website at http://support.avaya.com for the most up-to-date documentation, product notices, and knowledge articles. You can also search for release notes, downloads, and resolutions to issues. Use the online service request system to create a service request. Chat with live agents to get answers to questions, or request an agent to connect you to a support team if an issue requires additional expertise.

Chapter 2: Overview

Introduction

The HDX Programmer's Guide provides information for developing applications that use the HDX API. The HDX API allows the exchange of data between AACC/ACCS flow/script and a third-party host application.



Using HDX, third-party applications can implement additional contact center services. The guide describes procedures for using the HDX API to develop third-party host applications. It documents the functions and data structures available from the library to communicate with a contact center script.

The AACC/ACCS HDX server runs on AACC/ACCS and facilitates communication between scripts and provider applications. After the application registers with the server, the system routes SEND INFO and SEND REQUEST commands to the provider application. The HDX server uses specific communication services to connect and exchange messages with a third-party application. Under the HDX API functions, a communications layer establishes and maintains the communications connection with the HDX server.

When security is enabled in AACC/ACCS, the HDX CORBA interface supports secure communication with client applications. Security is enabled in AACC/ACCS using Security Manager. Consult AACC/ACCS documentation for information related to security in AACC/ACCS.

When security is enabled, the HDX CORBA interface supports both secure and unsecured communication. Existing unsecured HDX CORBA client applications continue to work even when security is enabled in AACC/ACCS. When security has been enabled, secure and unsecured HDX CORBA client applications can connect simultaneously to AACC/ACCS.

AACC/ACCS HDX CORBA uses mutual TLS to authenticate the identity of CORBA client applications. AACC/ACCS HDX CORBA is configured to use TLS v1.2.

HDX was originally designed for the voice contact type but is applicable to all contact types including multimedia contacts. The scripting commands for HDX are fully applicable for multimedia contacts.
HDX provides support for Common Object Request Broker Architecture (CORBA). CORBA offers programming language, network, and platform independence. Client provider applications can use the language of choice (such as C++ or Java) and the deployment platform of choice (such as Linux or Windows) to implement the solution.

The HDX server implementation is fully CORBA 2.0 compliant. Third-party HDX client applications that are not fully CORBA 2.0 compliant, or that use proprietary Object Request Broker (ORB) extensions, may not work correctly when connected to the HDX server.

Typical Usage Scenario

With the HDX API, third-party applications can provide services to callers through a script/flow. You can use the script commands SEND INFO, SEND REQUEST / GET RESPONSE to influence the script operation.

The following example uses a voice session to retrieve an account number from the caller. A HDX session sends the account number to the third-party application and retrieves the customer's account type. The call is then routed to a skillset appropriate to the account type. AACC/ACCS provides a set of API functions that an application can use to provide services. HDX supports multiple applications residing on one or more client hosts.

```
OPEN VOICE SESSION
      PLAY PROMPT Enter_Your_Acct_Num
      COLLECT DIGITS 10 INTO Caller_Acct_Num
END VOICE SESSION
ASSIGN 2 to Acct_Type
SEND REQUEST ProviderID Caller_Acct_Num
GET RESPONSE ProviderID Acct_Type
WHERE Acct_Type EQUALS
  VALUE 1: QUEUE TO SKILLSET MOST IDLE AGENT
  VALUE 2: QUEUE TO SKILLSET Standard_Group
  VALUE 3: QUEUE TO SKILLSET Overdue_Accts
END WHERE
```

Scripts can send data, as well as query and receive responses from a provider application. Data passes between the script and the provider application by including variables as parameters in the script commands.

Call ID is an important concept when building a Contact Center applications. The Call ID is the reference number that uniquely identifies a call or contact over the lifetime of that call or contact. For telephony calls, the switch generates the Call ID. For multimedia contacts, the Call ID is the external ID assigned by the multimedia provider when the contact is created.

The Call ID is sent automatically with every SEND INFO or SEND REQUEST command. You can use this Call ID so that information about the call (such as skillset or caller-entered data) can be used subsequently in a screen pop on the agent's desktop, if available.

Held Call ID is the Call ID of the call currently being transferred or conferenced. Held Call ID is populated for telephony calls only.

ProviderID

You must use an ID to uniquely identify an application. The value of the ProviderID must be available to both the provider application and the script writer. You must, therefore, be able to configure the ProviderID that is used by the provider application software, so that the ID is unique to a AACC/ACCS site.

In AACC/ACCS, applications can log in using the ProviderID of an already registered application. To use this feature, the Windows Registry location `AllowProviderReRegister` must be set to1. To turn off, set `AllowProviderReRegister` to 0.

`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Nortel\ICCM\HDX\`**`AllowProviderReRegister`**

Timing

Each request that is sent from a script to a service provider is time-stamped. If a completion message is not received from a service provider within the timeout period the HDX server discards the request and notifies the executing script of the timeout. The script continues without the expected data. You can specify the amount of time to wait for a response from the host to a maximum of 20 seconds.

Providing services within a script

You must perform the following two major activities for a third-party application to interact with an executing script:

1. Use HDX commands in your scripts.
2. Create the provider application.

The provider application has three options:

WIN32 API	- Support only WIN32 environment.

	- Delivered as a DLL that allows for client provider applications to be built in ANSI or Unicode and connects to the HDX server using RPC. - Goes through an internal proxy bridge to translate the Win32 RPC communications into CORBA.No TLS support - Unsecured communication only.
CORBA API	- Supports only WIN32 environment. - Delivered as a DLL that contains the same interface as the WIN32 API but connects to the HDX server using CORBA. - Only available for Unicode builds. - Unsecured communication only.
CORBA (recommended)	- Supports both non-Win32 and Win32 environments. - Develop a CORBA-based program using the provided IDL files, which can be compiled from the SDK using the TAO runtime and used to connect directly to the HDX server. - Supports secure communication. The client can use a secure connection to interact with the server.

API functions are used to register the application, send data, and retrieve data.

If you plan to release a Win32 provider application that does not include any of the HDX DLLs, then choose the CORBA implementation.

The CORBA interface is the native interface for the HDX implementation. The Win32 C interface goes through an internal proxy bridge to translate the Win32 C interface into the CORBA interface.

Both the WIN32 and CORBA provided DLLs share the same API interface and are interchangeable if the client provider application is built under Unicode. The WIN32 version uses remote procedure call (RPC), which is connected to the HDX server by way of a proxy, while the CORBA version is a direct connection to the HDX server.

Limitations

Limitations associated with the HDX interface.

1. You can specify a maximum of 10 parameters in the HDX commands. Each parameter is limited to 80 characters.
2. All data is transmitted in a character representation.
3. Parameters in the HDX commands must be variables or intrinsics. For the GET RESPONSE command, the parameters must be call variables. Constant values are not accepted.
4. Parameters must be single items. Sets and ranges are not accepted.
5. Variable types for parameters must be DN Digit, AGENTID, CLID, ACD, CDN, DNIS, LOC, NPA, NXX, NPANXX, SKILLSET, integer, or string type. Other types are not accepted.
6. Intrinsics for parameters must be CDN, CLID, DIALED DN, DNIS, or CALL DATA 1 through CALL DATA 10. Other types are not accepted.

7. HDX supports a maximum of 10 provider applications. Each of the provider applications can be running on any machine.

Chapter 3: HDX SDK Installation

Introduction

Use the HDX SDK to write, build, and execute a provider application developed in accordance with the HDX API. The setup program provides the header files, IDL files, libraries and binaries that you need to develop and deploy a HDX provider application.

In addition, the HDX SDK includes a TAO development and deployment environment that the developer can use to quickly and easily develop third-party CORBA client provider applications.

The HDX SDK installs WIN32 and a CORBA-based sample provider application along with sample source code to develop client provider applications using WIN32 or CORBA.

HDX Run-time Environment

The HDX run-time environment allows for the execution of prebuild sample applications. A WIN32 ANSI, WIN32 Unicode, and CORBA runtime are supplied for testing.

To successfully execute an application, you need a connection to an operational AACC/ACCS.

Obtaining the HDX SDK

The HDX SDK is available on the Avaya developer's Web site www.avaya.com/devconnect.

To install the HDX SDK

1. Remove any existing HDX SDKs installed on your system using the instructions in the Programmer's Guide for the version of the SDK.
2. Download the HDX SDK from the developer's Web site onto the development server.
3. Run the setup program to install the HDX SDK.

Notes:

1. The HDX SDK utilites and applications require write access to the install location. Ensure the logged in user has sufficient privilege to allow write access.

2. The HDX SDK utilites and applications require the Visual Studio 2017 redistributable. The installer allows the user to automatically install the redistributable.

To uninstall the HDX SDK

From the Control Panel, remove **Avaya HDX SDK.**

Migrating older HDX API applications

An existing application compiled against the Win32 Symposium HDX API can be run against an AACC/ACCS without recompiling. However, the run-time environment from the current HDX SDK must be used by the application.

Chapter 4: Scripting support for HDX

Introduction

HDX script commands have a generic format that allows data type and number of parameters to vary. This flexibility allows you to define a format to suit the needs of the specific application service.

Variables

You must use variables for the parameters whenever HDX commands are used in a script. Variables, as defined in many programming languages, are containers for data. Variables can be one of several different data types, such as an integer, string, CDN, DN, CLID, DNIS, ACD, LOC, NPA, NAPNXX, NXX, AGENTID, and SKILLSET. The type of data contained within the variable must agree with the variable type.

Scripts can contain two classes of variables: global variables and call variables. Global variables are read-only and cannot be modified by the executing script. They are shared among calls and scripts. Each call has its own set of call variables, the contents of which can be changed by the execution of a script.

Intrinsics

Intrinsic variable types are read-only and cannot be modified by the executing script. They are created and maintained automatically by the server and can be accessed throughout the system in any scripts. Intrinsics can be one of several different variable types, such as a CDN, CLID, Dialed DN, or DNIS.

HDX script commands

AACC/ACCS Orchestration Designer provides commands for communicating with a HDX application. Consult the AACC/ACCS document *Using Contact Center Orchestration Designer*.

SEND INFO	Sends data to the application. No response from the application is expected.
SEND REQUEST	Requests service from the application. A response is expected from the application and is retrieved through the Get Response command.
GET RESPONSE	Retrieves a response from the application, which is initiated by the SEND REQUEST command.

SEND INFO

```
SEND INFO <ProviderID> <P1>, <P2>, ..., <Pmax>
```

The system uses the SEND INFO command to send data to the host application. The data is stored in one or more parameters, which can be call variables, global variables, or intrinsics. Constant values are not allowed. The script does not expect a response from this command. The maximum number of parameters allowed in a single statement is 10.

Example: Sending a student record query
Each student record contains fields for student name, student ID, subject, and final grade.

```
// comments - variables:
// ProviderID = 12345
// type = student record
// name = Joe Smith;
// ID = 8515;
// subject = Math;
// grade = A
....
SEND INFO ProviderID type, name, ID, subject, grade
....
```

Note: Parameters following ProviderID are separated by commas.

The ProviderID identifies the service-providing application to which information is being sent. The ProviderID must have the same value that the application specifies when it registers with the HDX server.

Individual data parameters are converted to their character representation and inserted into a data container for shipment to the application. When received, the application extracts the individual items and converts them to their proper data type. The HDX API library provides methods to insert and extract data items from the data container.

When the host application receives the data, it interprets Joe Smith as the student name, 8515 as the student ID, Math as the subject, and A as Joe's grade for Math.

You must specify the exact use of this script command so that it can interwork with the provider application. The number of parameters, the content, and the order of the parameters determine how data is presented to the provider application.

Note: If a provider application has more than one service to provide, then one of the parameters can be a command that indicates the type of service requested.

SEND REQUEST

```
SEND REQUEST <ProviderID> <P1>, <P2>,..., <Pmax>
```

The system uses the SEND REQUEST command to request specific data from the application. Parameters are used in this command to identify the data of interest. Parameters can include call variables, global variables, or intrinsics. Constant values are not allowed. A script expects a Get Response to follow a SEND REQUEST. The maximum number of parameters allowed in a single statement is 10.

Example: Sending a student record query

Send the student name to identify the record.

```
// comments - variables:
// ProviderID = 12345
// type = student record
// name = Joe Smith;
...
SEND REQUEST ProviderID type, name
GET RESPONSE......
...
```

The ProviderID identifies the service-providing application to which the request is being sent. Individual data parameters are converted to their character representation and inserted into the data container for shipment to the application. When received, the provider application extracts the individual items from the data container and converts them to their proper data type.

When the application receives data, it interprets Joe Smith as the student name, and returns Joe's student record in the response.

You must specify the exact use of this command so that it interworks with the provider application. The ProviderID must have the same value that the provider application specifies when it registers with the HDX server.

The number of parameters, the content, and the order of the parameters determine how the data is presented to the application when it receives the SEND REQUEST.

Note: If a provider application has more than one service to provide, then one of the parameters can be a command that indicates the type of service requested.

GET RESPONSE

```
GET RESPONSE <ProviderID> <P1>, <P2>,..., <Pmax>
```

The system uses the GET RESPONSE command to obtain a response that pertains to the SEND REQUEST command sent to the provider application. A script's validation fails unless the SEND REQUEST command precedes the GET RESPONSE command.

You can set the timer setting parameter to control the time to wait for a response. At run time, if there is no response within the time specified by a default timer in the script, then the command fails and is aborted. The script continues to execute the next line, and leaves the call variable values unchanged in the GET RESPONSE statement. The default timer in the script program is currently set at 10 seconds.

You can specify one or more parameters to hold data in the response message; the parameters must be call variables separated by commas. The maximum number of parameters allowed in a single statement is 10.

Note: For more information about the timer setting parameter, see the Scripting Guide.

Example: Receiving a student record

Receive a student record from the host.

```
// comments - variables as data holders:
// ProviderID: 12345
// name: to hold the student's name
// ID: to hold the student's ID
```

```
// subject: to hold the subject that the student has taken
// score: to hold the student's score for the above subject
...
SEND REQUEST ProviderID type, name
GET RESPONSE ProviderID name, ID, subject, score
...
WHERE score EQUALS
...
```

You must also specify the exact use of this command so that it can interwork with the provider application.

The ProviderID identifies the service-providing application from which information is being retrieved. The ProviderID parameter must have the same value that the application specifies when it registers with the HDX server.

The number of parameters, the content, and the order of the parameters must correspond to the format of the information that the provider application returns to the script after it services the request.

Individual data items must be converted to their character representation and inserted into the data container for shipment to the HDX server application. When received, the parameters are extracted and converted to their proper data types for assignment to the call variables representing that parameter in the GET RESPONSE command.

If no response is received, then the GET RESPONSE command times out, the script continues to execute, and the parameters remain unchanged by the call. You must provide default values for variables used in the GET RESPONSE command so that appropriate action can be taken within the script for a failed command.

Chapter 5: Creating a CORBA service-providing application

Introduction

CORBA based interfaces are provided by AACC/ACCS. CORBA offers programming language and platform independence. Client provider applications can use the language of choice (such as C++ or Java), and the platform of choice (such as Linux or Windows) to implement the solution.

Secure Communcations

When security is enabled in AACC/ACCS, the HDX CORBA interface supports secure communication with client applications. Security is enabled in AACC/ACCS using Security Manager. When HDX CORBA is starting with security enabled, the security certificates are automatically extracted from the keystore which has been populated by Security Manager.

In AACC/ACCS, the security configuration needed for HDX CORBA is specified in the file D:\Avaya\Contact Center\Manager Server\TAO17\conf\hdx.conf.

Unicode

The character set supported by the CORBA interfaces is Unicode. Unicode is a character set defined by the international standard ISO 10646. The set of characters defined by the standard represents almost all known languages, including mathematical and scientific symbols. The standard relies on 16-bit character encoding (instead of the 8-bit encoding defined by ASCII).

The character set in the AACC/ACCS CORBA interface is defined as type unsigned short (or wchar_t in a Linux or Win32 environment). Do not use the type char (8-bit) when programming to the CORBA interface.

As an example, development under the Linux system requires you to use the C type wchar_t for Unicode support. Library functions (in Linux) like wctomb and mbtowc can be used to translate between Unicode and the 8-bit character char (see the Linux manual page on Unicode for details).

Interface Definition Language file

Interface Definition Language (IDL) provides a standardized representation of an object—and its methods and attributes—that remains consistent regardless of language or operating system platform. The SDK includes IDL files that you can use in developing your CORBA applications.

When you run the IDL files through the IDL compiler for your target language, it generates stub code that is usable in your HDX SDK.

Logon

As part of the registration process, the client specifies a Unicode user name and password. The user name and password must be validated against allowable users. Validation failure, due to incorrect user name, incorrect password, or both, generates an authentication-failed exception.

HDX is version-controlled. The provider application must include the version as part of the logon procedure. The version information has a major and minor release number; both values are included in the IDL file. The provider application should use the major and minor release number found in the IDL file for the provider application to use the correct interface. If the HDX service version is compatible with the provider application version, the provider application can log on to the HDX service. Incorrect versions are raised with incompatible version exceptions that the provider application can catch.

HDX Messaging

A HDX message contains the message payload and information identifying the message. The payload is a two-dimensional array of wide characters. The size of the array is set to 10 by 80. The payload can contain 10 strings of 80 wide characters each. The identifying information contains information such as reference ID and call ID.

For more information, see the CORBA Interface Definition.

CORBA development environment

CORBA is an open standard specification that allows compatible ORBs to communicate with each other. Avaya uses TAO 2.5.1 for ORB communication. The ORB is configured to use the Internet Inter-ORB Protocol (IIOP) for communication.

The HDX CORBA interface adheres to Object Management Group's (OMG) CORBA 2 specification. Any ORB compliant with the CORBA 2 specification can interoperate with the HDX.

The HDX SDK samples were implemented and tested with TAO 2.5.1 running on Windows 10 using Visual Studio 2017.

The assignment of the TCP/IP port for the TAO Naming Server is controlled properties files located on the server at the following location:

```
D:\Avaya\Contact Center\Manager Server\TAO17\properties\hdx.ini
```

Note: This runtime and libraries are supplied as part of the HDX Software Development Kit.

Dedicating a listener port

The default operation for HDX is to allow TAO to randomly pick a listener port number. If you have security concerns, you may want to prevent the random assignment of the port number by dedicating a port number.

1. Edit the file `D:\Avaya\Contact Center\Manager Server\TAO17\properties\hdx.ini`. The dedicated port is written to the HDX_Port entry. A value 0 indicates a random port. Save.
2. Using SCMU, stop and AACC/ACCS.

To create the provider application

HDX SDK provides C++ and Java examples for both secure and non-secure connection to the server, but can be adapted for other languages. The example is based on a TAO 2.5.1 implementation. It requires access to TAO/ACE include files, libraries, and binaries.

1. Use the TAO IDL compiler to compile the IDL files into source code.

2. Write the main C++ program to retrieve a reference (using the Naming Service or IOR) to the HDX object and access the operations defined by the IDL interface.

3. Compile and link the provider application.
    a. For compilation, define the preprocessor options and include directories.
    b. For linking, define the dependent libraries.

4. Run the provider application. Ensure the client application has access to the run-time environment.

A typical provider application scenario

A provider application is required to interact with the AACC/ACCS flow/script. Typically, the provider application must perform the following steps:

1. Register with the ORB. For a multithreaded client application create threads for handling incoming requests sent by the HDX server.

2. The provider application registers itself with the HDX service DX_RegisterProvider(). As part of the registration request, it passes the user name, password, version, and provider ID.

3. The HDX service acknowledges the request by passing back the HDX message object (see “HDX message”).

4. By using the HDX message object, the provider application can request a message in either a block (sync) or non-blocking (async) mode, DX_GetMessage(). The non-blocking mode is essentially a polling mechanism.

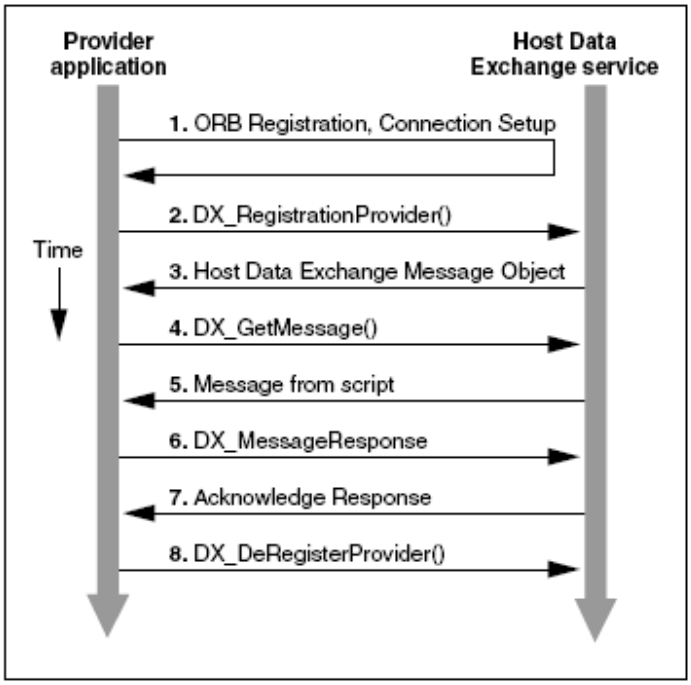
5. A message is sent using the Contact Center Manager script. The message arrives at the provider application, and the provider application acts on the information. In addition, the provider application should check to determine whether this message requires a response in which case the MsgType in the message is DXMT_ReqRespMsg.

6. If a response message is necessary, the provider application sends the response message, DX_MessageResponse(). The MsgReferenceID in the new message must be set to the MsgReferenceID of the received message.

7. The HDX service acknowledges the message response request.

8. The provider application shuts down by requesting a termination of service, DX_DeRegisterProvider(). The HDX message object is passed in.

Provider application—HDX interaction



Chapter 6: Creating a Win32 Service-Provider

To create a Win32 application

1. Use the DX_ProviderInit() function to initialize the HDX API library.

2. Use DX_RegisterProvider() to register the application as a service provider. The system sends a DXM_REGISTER_PROVIDER message to the HDX server. The message contains the provider ID and flag, which indicates a register operation.

3. Call DX_GetEvent() to retrieve incoming DXM_SERVICE_REQUEST, DXM_SERVICE_INFO, and DXM_RESPONSE messages.

4. With each retrieved message, extract the message type from the message data container and determine the type of action required:
    a. If the message is a response to a function call, such as DX_RegisterProvider(), then retrieve the result value from the message data to determine the status of the call.
    b. If the message is a DXM_SERVICE_REQUEST or DXM_SERVICE_INFO message, then the message data container contains the information sent with the message.
    c. If the message is a DXM_SERVICE_REQUEST, then retrieve the server ID from the request message. The server ID is a unique number assigned to the request by the HDX server and should be set into a service completion message.

5. If the message is a DXM_SERVICE REQUEST, after a service is completed, insert the results in the message data container and set the server ID obtained from the corresponding request. Then use the DX_SendServiceComplete() function to return the results to the caller. If the message is a DXM_SERVICE_INFO or DXM_RESPONSE message, then you do not need to perform this step. *Skip to step 6*. This function causes a DXM_SERVICE_COMPLETE message to be sent to the HDX server. The HDX server forwards the message to the originator of the request.

6. At shutdown, use the DX_DeRegisterProvider() function to deregister the application. This function sends a DXM_REGISTER_PROVIDER message with a flag set to deregister the application. When the application shuts down, it must call the DX_ProviderDeInit() function to allow the library to close its connections and free any held resources.

WIN32 DLL is used for ANSI and Unicode builds.

CORBA DLL may only be used for Unicode builds but provides a CORBA direct connection without internal proxies.

The API interfaces for the DLLs are all the same so the development of a client provider application is identical.

Chapter 7: HDX API library

Data Exchange API

Data types and structures

API functions use several different data types and structures as parameters and return values. The data is defined in the file `HDXSDK\WIN32\include\dx_types.h`.

clDX_Message_Data

clDX_Message_Data is a container for the information stored within a Data Exchange message. Almost all of the API functions require a handle to a clDX_Message_Data container as an argument. The definition of this data type is not provided to users of the library. You can access the clDX_Message_Data container through the API set provided by the HDX library.

DX_DATA_HANDLE_ TYPE

`DX_DATA_HANDLE_TYPE` serves as a handle to an object of the clDX_Message_Data type. You need an object of DX_DATA_HANDLE_TYPE for most of the API functions available in the Data Exchange library, including functions to insert and retrieve information from within the referenced clDX_Message_Data container.

DX_MESSAGE_TYPE

`DX_MESSAGE_TYPE` is an enumerated type used in the clDX_Message_Data container to specify the type of message.

```
typedef enum
{
       DXM_SERVICE_REQUEST,
       DXM_SERVICE_INFO,
       DXM_SERVICE_COMPLETE,
       DXM_RESPONSE,
       DXM_REGISTER_PROVIDER,
       DXM_ERROR,
       DXM_QUERY_STATUS,
       DXM_INVALID_MESSAGE
       DXM_SERVER_SHUTDOWN
}DX_MESSAGE_TYPE;
```

The enumeration contains all the message types used in the Data Exchange system. Not all of these messages are presented to an application that uses the API.

An application can expect to receive DXM_SERVICE_REQUEST, DXM_SERVICE_INFO, DXM_SERVER_SHUTDOWN, and DXM_RESPONSE messages through a call to the DX_GetEvent() function. DXM_SERVICE_COMPLETE and DXM_REGISTER_PROVIDER messages are sent to the HDX server when the API functions DX_SendServiceComplete(), DX_RegisterProvider(), and DX_DeRegisterProvider() are invoked.

DX_STATUS_TYPE

`DX_STATUS_TYPE` is an enumerated type used as the return value to specify the status of a particular call.

```
typedef enum
{
       DXS_UNKNOWNERR,
       DXS_SUCCESS,
       DXS_NO_EVENT,
       DXS_TIMEOUT,
       DXS_DLL_INIT_FAILS,
       DXS_COMM_INIT_FAILS,
       DXS_DATA_AVAILABLE,
       DXS_INCOMPATIBLE_VERSIONS,
       DXS_BAD_HANDLE,
       DXS_BAD_PARAM,
       DXS_QUERY_OK,
       DXS_REG_PROVIDER_EXISTS,
       DXS_NO_REG_PROVIDER_EXISTS,
       DXS_INVALID_PROVIDER_ID,
       DXS_PROVIDER_REGISTERED,
       DXS_PROVIDER_DEREGISTERED,
       DXS_PROVIDER_ID_EXISTS,
       DXS_PROVIDER_ID_NOT_FOUND,
       DXS_REGISTRATION_FAILED,
       DXS_ROGUE_WAVE_FAIL,
       DXS_DATA_SIZE_TOO_LONG,
       DXS_COMM_SEND_FAILS
}DX_STATUS_TYPE ;
```

DX_OPERATION_ MODE_TYPE

`DX_OPERATION_MODE_TYPE` is an enumerated type used in the DX_GetEvent() function to specify the blocking/non-blocking behavior of the call.

```
typedef enum
{
       DXO_SYNC,
       DXO_ASYNC
}DX_OPERATION_MODE_TYPE ;
```

Session Termination

If the connection between the HDX server application and the service-providing application is terminated, then the DXM_SERVER_SHUTDOWN message is generated and stored in the

message queue. When the service-providing application retrieves this shutdown message through the API functions, the service-providing application should take the appropriate actions, such as reestablishing the connection to the server.

HDX API Definitions

The HDX API defines data structures and API definitions to represent the interface. To be a Service Provider, an application must link with the HDX API library. This library encapsulates all of the communication software and procedures into a Dynamic Link Library (DLL) with a standard C functional interface. The following section describes the API functions and data types provided by this library.

The following API functions access message data in a clDX_Message_Data container with a DX_DATA_HANDLE_TYPE handle. The API is defined in the file `HDXSDK\WIN32\include\`**`dxprovid.h`**.

DX_createData Container()

DX_DATA_HANDLE_TYPE DX_createDataContainer();

This function creates a data container and returns its handle. The data container stores message information to be sent to the HDX server. After the message is sent, the data container is destroyed using API DX_destroyDataContainer() to free up its allocated memory.

Type	Value	Description
DX_DATA_HANDLE_TYPE	some memory address	Handle to the message data container created.
DX_DATA_HANDLE_TYPE	NULL	Failed to create a container.

DX_destroyData Container()

```
DX_STATUS_TYPE DX_destroyDataContainer (DX_DATA_HANDLE_TYPE
MessageDataHandle);
```

This function destroys the data container to which the parameter points. The function also frees the memory used by the data container.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description

DX_STATUS_TYPE	DXS_SUCCESS	Destroyed the data container.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid data container handle.

DX_getMessageType()

```
DX_MESSAGE_TYPE
DX_getMessageType(DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function returns the message type that corresponds to the handle passed as a parameter.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data handle.

Type	Value	Description
DX_MESSAGE_TYPE	DXM_SERVICE_INFO	Informational message.
DX_MESSAGE_TYPE	DXM_SERVICE_REQUEST	A service request message.
DX_MESSAGE_TYPE	DXM_SERVICE_COMPLETE	A service completion message.
DX_MESSAGE_TYPE	DXM_RESPONSE	A response message.
DX_MESSAGE_TYPE	DXM_INVALID_MESSAGE	Problem extracting message type from container.
DX_MESSAGE_TYPE	DXM_SERVER_SHUTDOWN	Session termination message.

DX_setMessageType()

```
DX_STATUS_TYPE DX_setMessageType (DX_DATA_HANDLE_TYPE
MessageDataHandle, DX_MESSAGE_TYPE MessageType);
```

This function sets a message type into the message that is sent to the HDX server.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.
MessageType	DX_MESSAGE_TYPE	A message type.

Type	Value	Description

DX_STATUS_TYPE	DXS_SUCCESS	Successfully set message type.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid message data handle passed in.

DX_getStatus()

```
DX_STATUS_TYPE DX_getStatus (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function returns the result value in a data container whose message type is DXM_RESPONSE.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to message data.

Type	Value	Description
DX_STATUS_TYPE	DXS_PROVIDER_REGISTERED	Successfully registered.
DX_STATUS_TYPE	DXS_PROVIDER_ID_ EXISTS	The provider ID is already in use.
DX_STATUS_TYPE	DXS_REGISTRATION_FAILED	A general failure in registration.
DX_STATUS_TYPE	DXS_PROVIDER_DEREGISTERED	Successfully deregistered.
DX_STATUS_TYPE	DXS_PROVIDER_ID_NOT_FOUND	The provider ID is invalid.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid message data handle passed in.

DX_setStatus()

```
DX_STATUS_TYPE DX_setStatus (DX_DATA_HANDLE_TYPE MessageDataHandle,
DX_STATUS_TYPE MessageStatus);
```

This function sets result status into the message data container.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.
MessageStatus	DX_MESSAGE_TYPE	Status

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Successfully set status into message.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid data container handle passed in.

DX_getServerID()

```
UINT32 DX_getServerID (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function retrieves the HDX server-assigned unique ID for the service request message. The function must set this unique ID into a return message, and send it to the HDX server so that the server can match the response with the request.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
UINT32	Some unsigned 32-bit integer	The unique ID of the service request message.

DX_setServerID()

```
DX_STATUS_TYPE DX_setServerID (DX_DATA_HANDLE_TYPE MessageDataHandle,
UINT32 nRefID);
```

This function sets a unique ID into a ServiceComplete message to be sent to the HDX server. The function must obtain this unique ID from the ServiceRequest message sent from the HDX server using API DX_getServerID().

The HDX server uses this ID to match the response with the request. It is critical that a ServiceComplete message has this ID set before it is sent to the HDX server.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.
nRefID	UINT32	Unique ID (within the DX_Server) of a request.

Type	Value	Description

DX_STATUS_TYPE	DXS_SUCCESS	Successfully set ID into message.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid data container handle passed in.

DX_getFirstItem()

```
TCHAR * DX_getFirstItem (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function returns the first parameter contained within the message.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
TCHAR *	-	String containing parameter or NULL if no parameters.

DX_setFirstItem()

```
DX_STATUS_TYPE DX_setFirstItem(DX_DATA_HANDLE_TYPE MessageDataHandle,
TCHAR * Parameter);
```

This function prepares the message to insert data one parameter at a time. The function also inserts the first parameter into the message. This function is only used when the message type is DXM_SERVICE_COMPLETE.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.
Parameter	TCHAR *	String containing the item to be inserted.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Successfully set status into message.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid data container handle passed in.
DX_STATUS_TYPE	DXS_UNKNOWNERR	Undefined error.

DX_STATUS_TYPE	DXS_DATA_SIZE_TOO_LONG	Data size exceeds the allowable limit.

DX_getNextItem()

```
TCHAR * DX_getNextItem (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function returns the next parameter contained within the message. The DX_getFirstItem() function must be called prior to using this function.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
TCHAR *	-	String containing parameter or NULL if no parameters.

DX_setNextItem()

```
DX_STATUS_TYPE DX_setNextItem (DX_DATA_HANDLE_TYPE MessageDataHandle,
TCHAR * Parameter);
```

This function inserts the next parameter into the message. The DX_setFirstItem() function must be called before this function can be invoked.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.
Parameter	TCHAR *	String containing the item to be inserted.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Successfully set status into message.
DX_STATUS_TYPE	DXS_BAD_HANDLE	Invalid data container handle passed in.
DX_STATUS_TYPE	DXS_UNKNOWNERR	Undefined error.
DX_STATUS_TYPE	DXS_DATA_SIZE_TOO_LONG	Data size exceeds the allowable limit.

DX_getCallID()

```
UINT32 DX_getCallID (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function retrieves the Call ID from the message. The Call ID identifies the call being processed by the script.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
UINT32	Some unsigned 32-bit integer	Call ID of the call requesting service.

DX_getHeldCallID()

```
UINT32 DX_getHeldCallID (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function retrieves the HeldCall ID from the message. The HeldCall ID is relevant to a call arriving in the call center and receiving script commands. The HeldCall ID applies only to telephony calls.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
UINT32	Some unsigned 32-bit integer	HeldCall ID relating to the call requesting service.

DX_getNetwork CallID()

```
UINT32 DX_getNetworkCallID (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function retrieves the NetworkCall ID from the message. The NetworkCall ID is relevant to a call arriving in the call center and receiving script commands. The NetworkCall ID applies only to telephony calls.

Parameter	Value	Description

MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
UINT32	Some unsigned 32-bit integer	Network call ID relating to the call requesting service.

DX_getNodeID()

```
UINT32 DX_getNodeID (DX_DATA_HANDLE_TYPE MessageDataHandle);
```

This function retrieves the Node ID from the message. The Node ID is relevant to a call arriving in the call center and receiving script commands. The Node ID applies only to telephony calls.

Parameter	Value	Description
MessageDataHandle	DX_DATA_HANDLE_TYPE	Handle to a message data container.

Type	Value	Description
UINT32	Some unsigned 32-bit integer	Node ID relating to the call requesting service.

DX_ProviderInit()

```
DX_STATUS_TYPE DX_ProviderInit (const unsigned long versionNum, const
TCHAR *providerSiteIP, const TCHAR *serverSiteIP, const TCHAR *Instance)
```

This function initializes the DLL for operation and establishes a communication connection with the HDX server.

Parameter	Value	Description
versionNum	const unsigned long	Version number that the API DLL uses.
providerSiteIP	const TCHAR *	IP address of the application service provider.
serverSiteIP	const TCHAR *	IP address of the HDX server.
Instance	const TCHAR *	Instance string of the provider.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	The API DLL is initialized.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	API DLL initialization failed.
DX_STATUS_TYPE	DXS_INCOMPATIBLE_VERSIONS	Invalid version number.

DX_RegisterProvider()

```
DX_STATUS_TYPE DX_RegisterProvider (DX_PROVIDER_ID_TYPE ProviderID);
```

This function registers the caller as a service provider. The ProviderID parameter is the ID of this provider if registration is successful.

Parameter	Value	Description
ProviderID	DX_PROVIDER_ID_TYPE	Unique ID of the service provider.

Type	Value	Description
DX_STATUS_TYPE	DXS_UNKNOWNERR	Undefined error.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	Initialization failed.
DX_STATUS_TYPE	DXS_SUCCESS	The registration message was sent (does not mean registered).
DX_STATUS_TYPE	DXS_REG_PROVIDER_EXISTS	The provider already registered.
DX_STATUS_TYPE	DXS_INVALID_PROVIDER_ID	Invalid provider ID.

DX_GetEvent()

```
DX_STATUS_TYPE DX_GetEvent (DX_OPERATION_MODE_TYPE mode, DX_DATA_HANDLE_TYPE * hdata);
```

This function fills in the data container with the information stored in the received message. Themode parameter specifies whether this function blocks.

A value of DXO_SYNC indicates that the call should block while it waits for a message. A value of DXO_ASYNC indicates that the call should return immediately, even if no messages are waiting.

The function retrieves one message (if one is available) from the queue of messages. The message data is inserted into the data container whose handle is passed in as a parameter.

The DXPROVID.DLL queues all received messages. At any given time, there can be several

messages waiting in the queue. If the calling application can process several messages at one time (for example, multithreaded), then it can call DX_GetEvent() multiple times to retrieve the waiting messages.

Parameter	Value	Description
mode	DX_OPERATION_MODE_TYPE	The mode of this operation (DXO_SYNC, DXO_ASYNC).
hdata	DX_DATA_HANDLE_TYPE *	Address of the handle to the message data container.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Retrieved an event into hdata.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	No message waiting.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	Initialization failed.

DX_Response Received()

```
DX_STATUS_TYPE DX_ResponseReceived (DX_DATA_HANDLE_TYPE * hdata);
```

An application calls this function when it has completed processing a DXM_RESPONSE message from a DX_GetEvent(). For example, this can be a response to the DX_RegisterProvider() function invoked earlier. The application uses this function simply to delete the data container.

The hdata parameter must refer to the same data handle used in the DX_GetEvent() function that retrieved the Response message. This function frees the resources of the message data container.

Parameter	Value	Description
hdata	DX_DATA_HANDLE_TYPE *	Address of the handle to the message data container (it must be the same handle used in the DX_GetEvent() function).

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Node ID relating to the call requesting service.
DX_STATUS_TYPE	DXS_BAD_HANDLE	The input data handle was invalid.

DX_InfoReceived()

```
DX_STATUS_TYPE DX_InfoReceived (DX_DATA_HANDLE_TYPE * hdata);
```

An application calls this function when it completes processing a DXM_SERVICE_INFO message received from DX_GetEvent(). This function is used to delete the data container.

The data parameter must be the same handle obtained from the DX_GetEvent() function that retrieved the DXM_SERVICE_INFO message. This function frees the resources of the message data container.

Parameter	Value	Description
hdata	DX_DATA_HANDLE_TYPE *	Address of the handle to the message data container. It must be the same handle used in the DX_GetEvent() function.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	The data container was deleted.
DX_STATUS_TYPE	DXS_BAD_HANDLE	The input data handle was invalid.

DX_SendService Complete()

```
DX_STATUS_TYPE DX_SendServiceComplete (DX_DATA_HANDLE_TYPE * hdata);
```

An application calls this function when it completes processing a request and wants to send a ServiceComplete message with appropriate data loaded back to the HDX server. The results must be packaged into the data container referenced by the hdata parameter. This function sends the message to the HDX server.

The HDX server forwards the service completion data to the originator of the service request.

Parameter	Value	Description
hdata	DX_DATA_HANDLE_TYPE *	Address of the handle to the message data container.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	The message was sent.
DX_STATUS_TYPE	DXS_BAD_HANDLE	The input data handle was invalid.
DX_STATUS_TYPE	DXS_UNKNOWNERR	Undefined error.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	Initialization failed.
DX_STATUS_TYPE	DXS_DATA_SIZE_TOO_LONG	One of the items in the data container exceeds the allowable limit.

DX_STATUS_TYPE	DXS_COMM_SEND_FAILS	Failed to send the message data to the server.

DX_DeRegister Provider()

```
DX_STATUS_TYPE DX_DeRegisterProvider();
```

This function deregisters the application as a service provider. No parameter is passed to this function.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	The message was sent (this does not mean deregistration).
DX_STATUS_TYPE	DXS_NO_REG_ PROVIDER_EXISTS	The provider is not registered.
DX_STATUS_TYPE	DXS_UNKNOWNERR	Undefined error.
DX_STATUS_TYPE	DXS_DLL_INIT_FAILS	Initialization failed.

DX_ProviderDeInit()

```
DX_STATUS_TYPE DX_ProviderDeInit();
```

This function frees all the resources held by the DLL and disconnects the communicationconnection with the HDX server.

Parameters are unnecessary for this function. The function should be called when the serviceprovider application ends operation.

Type	Value	Description
DX_STATUS_TYPE	DXS_SUCCESS	Successful cleanup.

Data formats

When data is sent from a script command to an application, the parameters for the command are converted to their character representation. The parameters are then packed into a clDX_Message_Data container pointed to by a DX_DATA_HANDLE_TYPE data handle for transmission. Parameters are placed in the container in the order in which they appear in the SEND INFO or SEND REQUEST command.

Note: The Call ID is an important concept when building a CTI application. It is used to tie data associated with the call to the voice call. The Call ID is sent automatically with every SEND INFO or SEND REQUEST command. You can use this Call ID so that information about the call (such as skillset or caller-entered data) can be used subsequently in a screen pop on the agent's desktop.

When the application uses the DX_GetEvent() function to retrieve the message, the container fills with message data that contains information about the request. The container's data handle is DX_DATA_HANDLE_TYPE and is passed in as a parameter to the function.

The parsing routines DX_getFirstItem() and DX_getNextItem() extract each parameter, still in character representation. The application later converts each parameter back into its appropriate type.

Before an application returns data using the DX_SendServiceComplete() function, the return data must be converted into character representation and packaged into a clDX_Message_Data container.

The order of the parameters in the clDX_Message_Data container and the types of data represented must agree with the format of the parameters of the GET RESPONSE script command. You must appropriately document the format of the parameters.

The functions DX_setFirstItem() and DX_setNextItem() are provided to load the clDX_Message_Data container with data through its data handle DX_DATA_HANDLE_TYPE.

Chapter 8: Service Provider Example

The samples installed as part of the HDX SDK installation show how an application uses the HDX API DLL. The examples are for guidance only. Customers should develop their own applications based on their specific needs.

Compiling and Linking a Service Provider

A service provided must be built to include HDX API header and library files. These are managed as compiler and linker options.

Compiler	Preprocessor	None
	Include Directories	Required include files: **1.** DX_Types.h **2.** DXProvid.h
Linker	Libraries	Required library: **1.** DXProvid.lib The library is located in the lib folder for the particular build flavour.

General Structure of a Service Provider

The general structure of a Service Provider application is initialization, processing and de-initialization.

Initialization / Registration	An application must first initialize the HDX API DLL and register with the HDX server as a service provider.
Receiving messages	After the initialization and registration, the HDX server sends a message to the application whenever a request for service is made by a script. When a message is received by calling the DX_GetEvent() function, the clDX_Message_Data container stores the message data. The clDX_Message_Data container's data handle, DX_DATA_HANDLE_TYPE, is passed as a parameter to the DX_GetEvent() function. Most applications call DX_GetEvent() periodically (through interrupts, WM_TIMER messages, polling, and so on) to check

	for messages and perform other duties when no message is pending. When a service request is received, the clDX_Message_Data container stores the data associated with the service request. This data is the information specified by the application software vendors in documentation for the services provided. Individual elements are in the order of their appearance in the script SEND REQUEST command. **Note**: Parameters are separated by commas in a script. ProviderID is not considered to be a parameter, but it is a necessary part of the command. The same is true for the script commands GET RESPONSE and SEND INFO.
Inserting data	When a service is performed and the results of the service are returned to the caller, those results must be inserted into the clDX_Message_Data container. This data is the information specified by the application software vendor's documentation. Individual elements must be in the order of their appearance in the script GET RESPONSE function.
Deregistration / De-initialization	When an application finishes providing services, it must deregister itself with the HDX server. This stops messages from being sent to the application. To complete the halt process, the HDX API DLL must also be deinitialized to allow the DLL to free any resources it may have acquired.

How to locate the HDX service

The HDX server attempts to register the following default CORBA compound name with the CORBA Naming service:

```
NortelNetworks\SymposiumCallCenterServer\HDX
```

In addition to the default CORBA compound name, each HDX server registers its Contact Center Manager Server site name with the Name Service in the manner shown:

```
NortelNetworks\SymposiumCallCenterServer\<Contact Center Manager server name>\HDX
```

By default, all HDX servers attempt to create and log the default CORBA name. In a network situation, only one HDX server can successfully register with the default CORBA compound name (the first); the other HDX server must register its Contact Center Manger site name. In a non-network environment, the client application can find the HDX server with the default CORBA compound name. Use the following procedure to locate the Naming Service.

Note: The utility `D:\Avaya\Contact Center\Manager Server\TAO17\bin\tao_nslist.exe` in AACC/ACCS lists the services registered with the Name Service.

The CORBA Naming Service location consisting of an IP address and port number is defined in the AACC/ACCS HDX.ini configuration file. The file is located

`D:\Avaya\Contact Center\Manager Server\TAO17\properties\hdx.ini`

The following is a sample HDX.ini file:

```
[TAO_Setup]
NameServerPort=4422
HDX_Port=0
ORBDebug=true
ORBDebugLevel=0
ORBSvcConf=
IORFile=
```

The AACC/ACCS HDX service automatically generates a new persistent IOR file when the service starts. The IOR file is called `C:\NIDXServer_ior.ref`

Chapter 9: Building a CORBA Application using TAO

Introduction to CORBA

CORBA (http://www.corba.org) is an architecture and specification for creating, distributing, and managing distributed program objects in a network. It allows programs located in different locations and developed by different vendors to communicate in a network through an interface broker. CORBA was developed under the auspices of the Object Management Group (OMG). It was designed to provide platform- and language-independent, object-oriented distributed computing.

The character set supported by the CORBA interfaces is Unicode (ISO 10646). The standard relies on 16-bit character encoding (instead of the 8-bit encoding defined by ASCII). The character set in the AACC/ACCS CORBA interfaces is defined as type unsigned short (or wchar_t in a Linux or Win32 environment). The type char (8-bit) is not supported on the CORBA interfaces.

Introduction to TAO

TAO (The ACE ORB, http://www.cs.wustl.edu/~schmidt/TAO.html) is an open source, CORBA-compliant, C++ Object Request Broker (ORB). TAO supports IIOP 1.2 enabling a high degree of interoperability with other conforming ORBs. It is implemented on top of ACE, which is infrastructure middleware that implements the core concurrency and distribution patterns for communication software. ACE is a highly portable, multiplatform framework that spans both real-time and general purpose operating systems. TAO uses ACE's high-performance, small footprint operating system adaptation layer for all operating system access, rather than invoking non-portable system calls directly. This allows TAO to be platform independent and easily ported to different operating systems.

Introduction to TAO Security

TAO provides an IIOP over SSL implementation called SSLIOP. SSLIOP can be used to enforce integrity, confidentiality and secure invocation when issuing client requests. Furthermore, it also provides the hooks by which X.509 certificate-based request authorization can be implemented in application code.

TAO's SSLIOP pluggable protocol implementation supports both the standard IIOP transport protocol and the secure IIOP over SSL transport protocol. As SSLIOP is implemeneted as a pluggable protocol, it is dynamically loaded into the ORB.

Basic ORB operation and communication

A client ORB communicates with a server ORB to deliver client request messages to the server and return responses from the server (if any) to the client. On the server, the ORB core delivers the requests to the appropriate Object Adapter and returns a reply message to the client-side ORB. The ORB also actively manages the transport-level communications that are used to transmit the requests and reply messages. As part of the OMG standards, a General Inter ORB Protocol (GIOP) is defined for enabling interoperable communications among disparate ORB implementations.

Basic CORBA client operation

A CORBA client application can access remote objects. To do this, it must obtain object references to the CORBA objects that it wants to access. The client can use a CORBA Naming Service or an IOR to obtain a reference to an object on the server. With a valid reference, the client can invoke operations on the object references. The CORBA client is unaware of how the CORBA object is implemented, and the only operations that are available to the client object are those defined in the objects interface, the IDL file. Note that each CORBA object has a unique identity and interface defined in the IDL.

Obtaining TAO

TAO can be downloaded from: http://download.dre.vanderbilt.edu/previous_versions/

The downloaded source must be built for the target platform. Build instructions are located at:

http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/TAO-INSTALL.html

Alternatively, you can obtain supported distribution of TAO from vendors. The current vendor

list is located at: http://www.cs.wustl.edu/~schmidt/commercial-support.html

Using the TAO IDL compiler to generate source code from the IDL

To use IDL interfaces, the IDL compiler is used to generate skeleton and stub code so requests can traverse from the client to the server. The code generated by the compiler maps is in accordance to standards set by OMG. The generated output files are not interchangeable between ORB implementations—files generated by the TAO ORB cannot be used by another ORB. The files must be compiled with the IDL compiler for the other ORB. However, the server and client can have two different ORB implementations. Therefore, while AACC/ACCS uses the TAO ORB, a third-party application can be built and deployed with an ORB from another open source or commercial vendor.

The third-party developer uses the TAO IDL compiler to generate source code from the IDL file. The source is compiled and linked to the third-party application.

IOR (Interoperable Object Reference)

Each of the AACC/ACCS services that implements a CORBA interface produces an IOR file. An IOR is a stringified object reference that is written to a file and allows objects to communicate across process boundaries. The IOR file is a data structure specified in the OMG CORBA 2.0 Interoperability specification. The IOR provides platform-independent and vendor-independent object references. The IOR is accessed by the client applications to obtain a reference to the server object. The IOR is useful in shared file systems; for example, the client application has access to the location of the IOR generated by the server.

The IOR files for AACC/ACCS are located at the root of the C: drive.

Naming Service

The Naming Service is a CORBA service that runs on the server. It allows CORBA objects to be named by means of binding a name to an object reference. The name binding is stored in the Naming Service. The client supplies the name to the Naming Service to obtain the reference to the desired object.

When security is enabled in AACC/ACCS, the HDX CORBA interface supports secure communication with client applications. However, the Naming Service is not secured. HDX CORBA client applications must connect unsecured to the Naming Service. See the sample code SampleClientTLS in the HDX SDK.

The TAO Naming Service in AACC/ACCS is configured with the following options:

m1	Multicast enabled. Clients can use IP multicast to query for a Naming Service, and this instance will respond. TAO Naming Server is listening for client multicast requests on a specified port. On the client side, <resolve_initial_references> sends out a multicast request on the network, trying to locate a Naming Service. When a Naming Server receives a multicast request from a client, it replies. The default multicast port is used.
ORBEndPoint iiop://[host]:4422	Specifies that the IIOP protocol is being used. The Naming Service is located on the host and listening on port 4422.
o tao_name_service.ior	Identifies the name of the file used to store the IOR of the root Naming Service context.

Reference persistence

All AACC/ACCS services using CORBA as the underlying architecture provide persistent references. The persistent reference allows the client to continue using a server reference even if the server is restarted.

The TAO Naming Service IOR is stored in the file.
`C:\Windows\SysWOW64\tao_name_service.ior`

This enables the client program running on another machine (not the local host) to copy the file to a directory (for example, D:\Name\) and can thus connect to the name service without searching for it using the connection reference

```
-ORBInitRef NameService=file:///D:\Name\tao_name_service.ior
```

TAO utilities

A number of TAO utilities is provided on AACC/ACCS to allow configuration and viewing of the Name Service. The utilities are located in `D:\Avaya\Contact Center\Manager Server\TAO17\bin`

The utilities are:

tao_nslist	Console Naming Service entries viewer.
NamingViewer	GUI Naming Service entries viewer.
tao_catior	Console IOR viewer.

Properties file

The ORB initialization options are configurable via the properties files
`D:\Avaya\Contact Center\Manager Server\TAO17\properties\hdx.ini.`

The properties file settings (default values shown) are:

```
NameService=iioploc://<ipaddress>:<NameServerPort>/NameService
NameServerPort=4422
iiop://<ipaddress>:<HDX_Port>
HDX_Port=0
ORBDebug=true
ORBDebugLevel=0
ORBSvcConf=
IORFile=
```

NameServerPort allows for the changing of the Naming Service port. It specifies the location of the Name Service; port is 4422 on the local machine. For more information about HDX.ini file details, see “To locate the CORBA Naming Service”.

When ORBDebug is set to true and the ORBDebugLevel is greater than 0 (max of 10), the TAO logging feature is activated. The log file is located in
`D:\Avaya\Logs\CCMS\*_OrbLog*.log`

The ORBSvcConf option allows the use of default configurations for the services.

TAO configures itself using the ACE Service Configurator framework. Thus, options are specified in the familiar svc.conf file (if you want to use a different file name, use the - ORBSvcConf option).

The IORFile option allows you to change the name of the default *.ior file produced by the service. Read the *.ini file to see what other services will be affected by this change.

Client-side settings for TAO

A TAO client is a CORBA application that actively establishes connections, submits requests, and receives responses from a TAO server. You must be careful when specifying the behavior of clients for multithreaded applications.

In particular, it is important to direct the Service Configurator behavior to provide exclusive access to the Transport so that requests are not multiplexed on a connection. You can use -ORBSvcConfDirective statis client-strategy-Factory "-ORBTransportMuxStrategyEXCLUSIVE" to prevent a multithreaded application from blocking. The default operation is to send and receive information on the same connection.

Note: A new ORB is created only for each thread. If a single-threaded application creates more than one ORB (using ORB_init()), it always references to the first ORB created for that particular thread.

For secure communication, the client must provide a service configurator file client.conf file with the configuration for SSLIOP pluggable protocol.

```
# client.conf
dynamic SSLIOP_Factory Service_Object * TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory()
       "-SSLAuthenticate SERVER_AND_CLIENT -SSLPrivateKey PEM:client_key.pem -
SSLCertificate PEM:client_cert.pem -SSLCAfile PEM:cacert.pem -SSLVersionList TLSv1.2"

static Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

Chapter 10: CORBA Interface Definition

The HDX SDK provides two interface definition files NIDXMessage.idl and NIDXProvider.idl. NIDXMessage.idl defines the structure of the message and NIDXProvider.idl defines the transport mechanism for the message.

NIDXMessage.idl

The message is composed of the following fields:

Message	Definition
Call ID	The Call ID identifies the call being processed by the script. For multimedia contacts, this field contains the contact ID of the multimedia contact.
Node ID	The Node ID is the ID of the DMS switch system. The field applies only to telephony calls..
Held Call ID	The Held Call ID is the Call ID of the call currently being transferred or conferenced. The field applies only to telephony calls.
Reserved Call Reference	The internal HDX reference.
Msg Reference ID	The HDX service assigns a unique ID for the service request message. This unique ID must be set into a return message and sent to the HDX service so that the service can match the response with the request.
Msg Provider ID	The provider ID that the message is destined for, or the provider of the response message.
Msg Application ID	The application sending the message.

Msg Type (the type of message sent):

Info Msg	Information message. A response to this message is not required.
Req Resp Msg	A request/response message.
Resp Msg	A response message.

Message	Definition
Timeout	A timeout message. This message is sent when the provider application does not respond in time when a request/response message is sent. Provider applications do not receive this type of message.
Time	The time when the message was created and sent. The time is represented in UNIX format (the number of seconds elapsed since midnight (00:00:00), January 1, 1970.

Info List	Contains the message information. The message information is limited to 80 characters with, at most, 10 unique messages.

NIDXProvider.idl

NIDXProvider.idl defines the functions required by the provider applications to connect and communicate with the HDX service. The IDL file defines the following functions:

Provider object

```
NIDXProvider::DX_RegisterProvider()
```

This function registers the caller as a provider application. The providerID parameter is the ID of this provider if registration is successful. This function returns the HDX provider message object.

Parameter	Description
providerID	The unique ID of the provider application.
UserID	The Unicode logon user name and password.
Version	The provider application member function version of the HDX interface. The version is provided in the IDL file.

```
NIDXProvider::DX_DeRegisterProvider()
```

This function deregisters the application as a provider application.

Parameter	Description
MsgObject	The HDX provider message object to be deregistered.

Provider object exceptions

In addition to the standard exception-generated signals (such as Communication Failure), the following signals are Avaya-specific:

- IncompatibleVersion—This exception is raised when a provider application has a version that is not supported by the HDX service.
- TooManyConnections—This exception is raised during registration when the HDX service's maximum provider applications limit has been reached.
- CurrentlyRegistered—This exception is raised during registration when a registered provider application attempts to reregister.
- AuthenticationFailed—This exception is raised when a provider application attempts an illegal logon (invalid user name, password, or both).
- InvalidObject—This exception is raised when a provider application attempts to deregister an invalid object.

Message Provider object

```
NIDXProviderMessaging::DX_GetMessage()
```

This function is called to retrieve the message sent by the service application (for example, the

script). A value of DXOM_SYNC indicates that the call should block, waiting for a message. A value of DXOM_ASYNC indicates that the call should return immediately, even if no messages are waiting. The function retrieves one message (if one is available) from the queue of messages.

Parameter	Type	Description
OpMode	DXOM_SYNC	Synchronous mode. The call is blocked until a message arrives.
OpMode	DXOM_ASYNC	Asynchronous mode. The call returns immediately even if no messages are waiting.
message	N/A	The message as defined in NIDXMessage.idl.

`NIDXProviderMessaging::DX_MessageResponse()`

This function is called by a provider application when it completes processing a request/response message from DX_GetMessage() and is sending the response message.

Parameter	Description
message	The message as defined in NIDXMessage.idl.

Chapter 11: Performance and Engineering

Because a script/flow is executed for each call, the performance of the service provider affects the performance of an executing script. For example, if the provider application can only handle one request at a time, and a request takes 4 seconds to complete, then AACC/ACCS only handles a maximum of 900 calls per hour.

You may want to provide applications with multiple request capabilities, or limit the services provided to shorten the time required to complete a single service.

Chapter 12: Testing HDX functionality

Overview

You can test the functionality of your HDX program by using the Provider tool. Provider uses the DXProvid.dll to communicate with HDX running in platform. You can use Provider to view the variable parameters sent from the HDX script functions, such as SEND INFO and SEND REQUEST. You can also use Provider to return the message data back to HDX as requested by the script function, such as GET RESPONSE.

Provider is included as part of AACC/ACCS and is located:

```
D:\Avaya\Contact Center\Manager Server\iccm\bin
```

The HDX SDK supplies MBCS and Unicode versions of the Provider application. Select the version to suit your operating systems. Most users will select Unicode.

To run the Provider application

1. Type **Provider** to start the application from the appropriate directory.
   Result: The Provider program main window appears.

2. Enter the appropriate IP addresses of the Provider and server locations in dotted format.
   Result: The default IP address is initialized to 127.0.0.1.

3. After you have entered the IP address, select **Connect to Server**.
   Result: You are connected to the server. An error dialog box appears if you type the IP addresses incorrectly.

4. In the Register ID box, enter a value. You must ensure that the value you enter in this box matches the application ID defined in the HDX script variable.

5. Click Register to register with the server.
   Result: After you click Register, the Register button changes to UnRegister if the registration is successful.

6. To unregister the application, click **UnRegister**.

To send data to Provider

1. Configure the script to send the message data to Provider using the HDX script commands. The providerid defined in the script variable must be the same value you entered in the **Register ID** box on the Provider screen. Sample script commands are:

   ```
   SEND REQUEST providerid g_dnis1
   GET RESPONSE providerid c_dnis1
   ```

```
SEND INFO providerid c_dnis1
```

2. Because the example requires Provider to return data back to the script, you must select the **Return parameter 1** option, and then enter the response digit numbers in the first completion data box. You should also select the **Automatic response** option so that Provider returns the message data immediately when receiving a request.
   Note: Click **Manual Response** when you want to return the message data manually. To use this feature, you must deselect the Automatic response option.

3. Make a call to execute the script. The Request data boxes are updated after the call is presented to an agent.

To return data manually

1. Configure the script to send the message data to Provider using the HDX script commands. The providerid defined in the script variable must be the same value you entered in the **Register ID** box on the Provider screen. Sample script commands are:

```
SEND REQUEST providerid TIMER 20 g_dnis1
GET RESPONSE providerid c_dnis1

SEND INFO providerid c_dnis1
```

2. Select the **Return parameter 1** option, and then enter the response digit numbers in the first completion data box. Ensure that the **Automatic response** option is not selected.
   Note: The timer setting allows the script to wait 20 seconds for the HDX application to reply.

3. Make a call to execute the script.

4. When the information is received, click **Manual Response** to return data that is stored in the first completion data box to the server.

LAST PAGE