objective
SYSTEMS, INC.

CSTA Phase 2 C++ API Evaluation Kit for Windows

User's Guide

Objective Systems, Inc.   June 2019

Introduction

The *Objective Systems' CSTA Phase 2 C++ API Evaluation Kit* for Windows is a collection of classes for encoding and decoding messages from the CSTA Phase 2 ASN.1 specification using the Basic Encoding Rules (BER) as defined in ITU standard X.690.

This API kit has been developed in the C++ programming language. The Objective Systems ASN1C compiler is used (by you, using supplied makefiles or project files) to generate the structures and encode/decode functions. These are then compiled (by you, using supplied makefiles or project files) using any of the compilers for which an ASN1C run-time collection exists. Since this kit is an evaluation kit, you must have a copy of the ASN1C software. If you don't have ASN1C, you can download a 30-day trial version at http://www.obj-sys.com/asn1-compiler.shtml.

Contents of the Package

The following diagram shows the directory tree structure that comprises the C++ CSTA phase 2 evaluation kit package:

```
csta2fw
 |
 +- build
 |
 +- build_dll
 |
 +- doc
 |
 +- lib
 |
 +- specs
 |
 +- src
 |
 +- sample
```

The purpose and contents of the various subdirectories are as follows:

- build: Contains the makefile to build the CSTA and ACSE runtime library for the CSTA phase 2 and ACSE source code API.
- build_dll: Contains the makefile to build the dll version of the CSTA and ACSE libraries.
- lib – Contains following libraries and DLLs.

cstap2_a.lib	Contains (after you build it) CSTA phase 2 protocol (ECMA-218) and ROSE protocol (ITU-T X.880 / ISO 13712-1) implementation in a static library.
acse_a.lib	Contains (after you build it) ACSE protocol (X.227 / ISO 8650) and related ASN.1 definition implementation in a static library.
cstap2.dll	Contains (after you build it) CSTA phase 2 protocol (ECMA-218) and ROSE protocol (ITU-T X.880 / ISO 13712-1) implementation in a DLL. There is also an accompanying cstap2.lib symbol table file.
acse.dll	Contains (after you build it) ACSE protocol (X.227 / ISO 8650) and related ASN.1 definition implementation in a DLL. There is also an accompanying acse.lib symbol table file.

- src – Contains the source code for the CSTA phase 2, ROSE, ACSE, Information Framework and UsefulDefinition ASN.1 specifications that are generated (by you) using the ASN1C compiler.

- doc – Contains this document

- specs – Contains the CSTA (ECMA-218), ROSE (X.880) , ACSE (X.227), Information Framework(X.501), and UsefulDefinitions (X.501) ASN.1 specifications that were used in the compilation.

- sample – Contains sample programs that illustrate how to use the API.

Getting Started

The package is delivered as a zip archive that should be unpacked into one of the cpp folders of your ASN1C installation. Which folder you choose depends on which compiler you want to use to build the software. The folder named just cpp is for the default version of Visual Studio for your version of the ASN1C SDK. You can determine which version of Visual Studio is the default by looking at the cpp_vs* folders in your SDK installation. The version of Visual Studio that is not mentioned in one of those folders is the default version. So if you want to use the default version of Visual Studio, you would unpack the zip into the cpp folder. If, on the other hand, you want to use a different version of Visual Studio, you would unpack the zip into the appropriate cpp_vs* folder. All makefiles and internal sample programs use relative directory paths, so it is not necessary to create any type of top-level environment variables.

The libraries must be compiled before use. It is necessary to have a working copy of ASN1C already installed on the system in order to generate the required source code. Running nmake in the build subdirectory will generate the source code, compile it, and package it into the library files described above. Alternatively, running nmake in the build_dll subdirectory will package the code into the DLL files described above You need to use a command window that is appropriate for the compiler you're using.

The code can be tested by executing the sample programs in the sample subdirectory. Most of these sample programs consist of a reader and writer program. The writer program populates a data variable with some data, calls an encode function, and then writes the encoded byte stream to a file. The reader program reads this file, decodes the data into a C++ structure, and then prints the decoded results.

The kit also includes Visual Studio 2010 solution and project files that build the libraries and DLLs as well as all of the samples. Only Visual Studio 2010 files are provided. If you're using a Microsoft compiler older than Visual Studio 2010, you must use the makefiles to do the building. If you're using a Microsoft compiler newer than Visual Studio 2010, the solution and project files should migrate forward if you open them in the IDE of the newer compiler.

CSTA Explicit Association

The CSTA protocol operates within an application association (otherwise known as a CSTA association or association) as provided by IS 8649 (ACSE). This association can be either:
- an implicit association achieved via an off-line agreement or
- an explicit association realized through the use of ACSE.

The initialization sequence of explicit associations is described in the following sections. Explicit/dynamic association can be realized by using acse_a.lib or acse.dll (ACSE ASN.1 implementation) API.

ACSE service or application context can be defined as follows:

- A-ASSOCIATE: This confirmed service is used to initiate an application association between application entities

    1. The A-ASSOCIATE service is initialized by sending a message of type *ASN1T_AARQ_apdu* and waiting for a response. To generate an AARQ-apdu message, the user will need to set the application context name to the CSTA Phase 2 object identifier value. The *ACSE-Request* sample program can be used as a reference.

    2. The response will be an *ASN1T_AARE_apdu* type. The responder replies with the accepted CSTA version for connection or reject reason. The protocol version to be used is selected by identifying the highest CSTA version that is common to both systems. The *ACSE-Response* sample program can be used as a reference.

- A-RELEASE: This confirmed service is used to release an application association between application entities without loss of information.

    1. The A- RELEASE service is initialized by sending a message of type *ASN1T_RLRQ_apdu* and waiting for a response. The sample program function *encodeACSEReleaseRequest()* in the *ACSE_Request* directory can be used as a reference to create the release request.

    2. The response will be a message of type *ASN1T_RLRE_apdu*. The sample program function *encodeACSEReleaseResponse()* in the *ACSE_Response* directory can be used as a reference to create the release request.

- A-ABORT: This unconfirmed service causes the abnormal release of an association with a possible loss of information.

    The A-ABORT service is initialized by sending a message of type *ASN1T_ABRT_apdu*. The sample program function *encodeACSEAbort()* in the *ACSE_Request* and *ACSE_Response* directory can be used as reference.

Encoding CSTA Messages with ROSE Header

The CSTA specification specifies a two-phase protocol using ROSE for the common headers. In order to encode a message of this type, the following steps must be performed:

1. A CSTA base message type must be encoded, and

2. The results must be plugged into a ROSE message structure and then this is encoded to produce the finished message.

The user should use the writer program (*writer.cpp*) in one of the sample directories as a guide when reading the rest of the procedure.

Encoding a CSTA message

To encode a CSTA message component, a variable of one of the various CSTA class structures must first be populated with data. These structures normally correspond to the ARGUMENT or RESULT types specified in a CSTA Phase 2 Information Objects for the OPERATION class. For example, the following information object specifies the messages that are exchanged for the *makeCall* operation:

```
makeCall OPERATION ::= {
      ARGUMENT    MakeCallArgument
      RESULT      MakeCallResult
      ERRORS      {universalFailure}
      CODE local : 10
}
```

In this information object, the CODE field defines the value "local: 10" to identify a **makeCall** operation. The name "local" is actually the name of a CHOICE in the ASN.1 specification that stipulates that the operation code will be an integer. The ARGUMENT field defines the *MakeCallArgument* type, which can be used to invoke the **makeCall** operation. The RESULT field defines the *MakeCallResult* type, which is used to return the result of the **makeCall** operation. The ERRORS field defines another information object that contains the error type that can be produced in a **makeCall** operation.

Table 1 is developed from the CSTA phase 2 OPERATION class information object definitions. This table contains the operation name, operation code, argument type and result type for each operation. To encode/invoke the operation, the user must set the operation code value and encode the argument type defined in this table. To decode the operation, the user must check the operation code value and decode the corresponding result type or argument type. For example, for the **makeCall** operation, the user will need to set the *ASN1T_CSTA_ROSE_PDU_invoke.operationCode* value to local:10 and encode the *ASN1T_MakeCallArgument* type in *ASN1T_CSTA_ROSE_PDU_invoke* argument open type field.

In this case, *MakeCallArgument* is encoded and sent as a request message (or invoke as it is known in ROSE). The entity receiving this message is then required to respond with the result message of *MakeCallResult* type or one of the defined errors in the `universalFailure` information object.

The sample program in the *makeCall_Request* directory shows how to encode a *MakeCallArgument*:

```
MakeCallArgument ::=
            SEQUENCE
            {callingDevice        DeviceID,
             calledDirectoryNumber  CalledDeviceID,
             extensions     CSTACommonArguments  OPTIONAL}
```

***Encoding "cSTAEventReport" Operations*:**

"cSTAEventReport" operation has the special case of event, which is different from other operations. To encode the event the user will need to encode the CSTAEventReportArgument type defined in the table 1; e.g, for the "***delivered***" event, user will need to set the *ASN1T_CSTAEventReportArgument* as follows.

The following is a snippet from the writer.cpp "delivered_Event" directory sample program showing how the value is populated:

```
/* delivered event */
ASN1T_DeliveredEvent delivered;
//populate value for delivered data structure

ASN1T_CallEvent callevent;
callevent.t = T_CallEvent_deliveredEvent;
callevent.u.deliveredEvent = &delivered;

/* CSTAEventReportArgument */
ASN1T_CSTAEventReportArgument eventReportArg;
ASN1OCTET data[] = { 0x99 };
eventReportArg.crossRefIdentifier.numocts = 1;
```

```
eventReportArg.crossRefIdentifier.data = data;
eventReportArg.eventSpecificInfo.t = T_EventSpecificInfo_callEvent;
eventReportArg.eventSpecificInfo.u.callEvent = &callevent;
```

.1   Encoding a ROSE Header

Once the argument is populated and encoded, the ROSE header must be added. This is a common header that is added to all messages that support the ROSE protocol. In the case of a ROSE OPERATION, a ROSE Invoke message must be sent to the other entity.

The ROSE header required to send an invoke message consists of 4 fields:

1. Invoke ID: this is an arbitrary identifier that acts as a "handle" for matching responses to requests when messages are exchanged. Any result or error received in response to this invoke request will contain this identifier value.

2. Linked ID: this is another Invoke ID that is used when a sub-operation within the existing operation is initiated. The Linked ID is the Invoke ID of the parent (i.e. the encapsulating) operation.

3. Operation Code: this identifies the operation to the receiving entity. Table 1 can be used to find out the operation code value for a particular operation. For example, the *makeCall* operation corresponds to the "local : 10" value.

4. Message Data: this is an open type. The CSTA encoded message data is placed in this open type field. Table 1 can be used to find out the type of message that should be used for a particular operation. For example, the *makeCall* operation corresponds to the *MakeCallArgument* type.

The following is a snippet from the writer.cpp sample program showing how the header is added:

```
/* Populate header structure */

invoke.m.argumentPresent = 1;
invoke.invokeId.t = T_InvokeId_present;
invoke.invokeId.u.present = 1; /* arbitrary number: should be unique */
invoke.opcode.t = T_Code_local;
invoke.opcode.u.local = 10; /* "makeCall" operation code */

/* This is where we get the previously encoded message component */
invoke.argument.numocts = msglen;
invoke.argument.data = (ASN1OCTET*) encodeBuffer.getMsgPtr();

pdu.t = T_CSTA_ROSE_PDU_invoke;
pdu.u.invoke = &invoke;
```

The header identifies the operation to be performed (opcode = 10 = makeCall) and assigns a unique invoke identifier. This invoke identifier serves as a session ID that can be used to match requests with responses if asynchronous communications are used. The last part of the populate logic gets the previously encoded message component from encoding the make call argument data. This is the open type onto which the ROSE header is prepended.

Decoding CSTA Messages

CSTA messages are decoded by reversing the procedure that was used to encode them. In other words, the following two distinct decode operations must be performed:

1. The ROSE header must be decoded, and

2. The CSTA message type must be decoded

This is the inverse of the encoding procedure presented earlier. The user should use the reader program (*reader.cpp*) in one of the samples as a guide when reading the rest of the procedure.

The procedure to decode a complete CSTA message is as follows:

1. Read an encoded message from an input stream.

2. Create an *ASN1BERDecodeBuffer* object to wrap the message buffer that the message was read into.

3. Create a *CSTA_ROSE_PDU* object and use it in conjunction with the decode buffer object created above to decode the header.

4. The header fields can now be examined. An application will first check Invoke ID to find out the response for different sessions. For our example, the value of the Invoke ID field should be "1", which is random unique number we have set during encode procedure. Then to identify the operation, the operation code value is checked. This should be equal to "local:10" for a result/error for our invoke request.

5. Create a second *ASN1BERDecodeBuffer* object using the open type data contained in the ROSE header structure as the message source.

6. Create the specific CSTA message type object based on the operation code value and corresponding result type from Table 1. Use this result type decode method to decode the CSTA message component.

Table 1: Operation Table for CSTA phase 2

Operation name	Operation Identifier	Operation Invoke type	Operation Result type
alternateCall	local: 1	AlternateCallArgument	AlternateCallResult
answerCall	local: 2	AnswerCallArgument	AnswerCallResult
callCompletion	local: 3	CallCompletionArgument	CallCompletionResult
clearCall	local: 4	ClearCallArgument	ClearCallResult
clearConnection	local: 5	ClearConnectionArgument	ClearConnectionResult
conferenceCall	local: 6	ConferenceCallArgument	ConferenceCallResult
consultationCall	local: 7	ConsultationCallArgument	ConsultationCallResult
divertCall	local: 8	DivertCallArgument	DivertCallResult
holdCall	local: 9	HoldCallArgument	HoldCallResult
makeCall	local: 10	MakeCallArgument	MakeCallResult
makePredictiveCall	local: 11	MakePredictiveCallArgument	MakePredictiveCallResult
queryDevice	local: 12	QueryDeviceArgument	QueryDeviceResult
reconnectCall	local: 13	ReconnectCallArgument	ReconnectCallResult
retrieveCall	local: 14	RetrieveCallArgument	RetrieveCallResult
setFeature	local: 15	SetFeatureArgument	SetFeatureResult

transferCall	local: 16	TransferCallArgument	TransferCallResult
associateData	local: 17	AssociateDataArgument	AssociateDataResult
parkCall	local: 18	ParkCallArgument	ParkCallResult
sendDTMFTones	local: 19	SendDTMFTonesArgument	SendDTMFTonesResult
singleStepConf	local: 20	SingleStepConfArgument	SingleStepConfResult
cSTAEventReport	local: 21	CSTAEventReportArgument	
routeRequest	local: 31	RouteRequestArgument	
reRouteRequest	local: 32	ReRouteRequestArgument	
routeSelectRequest	local: 33	RouteSelectRequestArgument	
routeUsedRequest	local: 34	RouteUsedRequestArgument	
routeEndRequest	local: 35	RouteEndRequestArgument	
singleStepTrans	local: 50	SingleStepTransArgument	SingleStepTransResult
escapeService	local: 51	EscapeServiceArgument	EscapeServiceResult
systemStatus	local: 52	SystemStatusArgument	SystemStatusResult
monitorStart	local: 71	MonitorStartArgument	MonitorStartResult
changeMonitorFilter	local: 72	ChangeMonitorFilterArgument	ChangeMonitorFilterResult
monitorStop	local: 73	MonitorStopArgument	MonitorStopResult
snapshotDevice	local: 74	SnapshotDeviceArgument	SnapshotDeviceResult
snapshotCall	local: 75	SnapshotCallArgument	SnapshotCallResult
startDataPath	local: 110	StartDataPathArgument	StartDataPathResult
stopDataPath	local: 111	StopDataPathArgument	StopDataPathResult
sendData	local: 112	SendDataArgument	SendDataResult
sendMulticastData	local: 113	SendMulticastDataArgument	SendMulticastDataResult
sendBroadcastData	local: 114	SendBroadcastDataArgument	SendBroadcastDataResult
suspendDataPath	local: 115	SuspendDataPathArgument	SuspendDataPathResult
dataPathSuspended	local: 116	DataPathSuspendedArgument	DataPathSuspendedResult
resumeDataPath	local: 117	ResumeDataPathArgument	ResumeDataPathResult
dataPathResumed	local: 118	DataPathResumedArgument	DataPathResumedResult
fastData	local: 119	FastDataArgument	FastDataResult
concatenateMessage	local: 500	ConcatenateMessageArgument	ConcatenateMessageResult
deleteMessage	local: 501	DeleteMessageArgument	DeleteMessageResult
playMessage	local: 502	PlayMessageArgument	PlayMessageResult
queryVoiceAttribute	local: 503	QueryVoiceAttributeArgument	QueryVoiceAttributeResult
reposition	local: 504	RepositionArgument	RepositionResult
resume	local: 505	ResumeArgument	ResumeResult
review	local: 506	ReviewArgument	ReviewResult
setVoiceAttribute	local: 507	SetVoiceAttributeArgument	SetVoiceAttributeResult
stop	local: 508	StopArgument	StopResult
suspend	local: 509	SuspendArgument	SuspendResult
synthesizeMessage	local: 510	SynthesizeMessageArgument	SynthesizeMessageResult
recordMessage	local: 511	RecordMessageArgument	RecordMessageResult

NOTE: For all the of the above operations or information objects, the return error type is *ASN1T_UniversalFailure*.

Common CSTA Operations

Making a Call

One possible need with CSTA messaging is to send the PBX a message telling it to make a call and then to retrieve the call id (a sequence of octets) for the resulting call from the response that the PBX sends back to the client.

The code sample below shows how this operation can be done. The sample uses a method called makeCall() within a class called CSTAEngine. The method accepts two arguments (the calling number and the number to call) and returns a pointer to the call id octet sequence. If anything goes wrong, the method simply returns NULL.

This code sample uses the following variable prefix conventions:

> psz – Pointer to null-terminated string.
> pach – Pointer to an array of chars (not necessarily null-terminated).
> t – Structure or object instance.
> pt – Pointer to a structure or object instance.
> i – Integer.

If a variable ends with _T, the variable refers to an ASN1C-generated data object. If a variable ends with _C, the variable refers to an ASN1C-generated control object.

```
#include "CSTA-call-connection-identifiers.h"
#include "CSTA-make-call.h"
#include "CSTA-device-identifiers.h"
#include "CSTA-ROSE-PDU-types.h"
#include "Remote-Operations-Information-Objects.h"

#include "asn1BerCppTypes.h"
#include "asn1CppTypes.h"
#include "ASN1TOctStr.h"

#include "CSTAEngine.h"

#include <memory.h>
```

These are the include directives that are necessary.

```
char *CSTAEngine::makeCall(char *pszCallingDevice, char
*pszCalledDevice)
{
    ASN1BEREncodeBuffer tEncodeBuffer;
    ASN1T_MakeCallArgument tMakeCallArgument_T;
    ASN1C_MakeCallArgument tMakeCallArgument_C(tEncodeBuffer,
tMakeCallArgument_T);
```

In this section the method is allocating an encode buffer, a data object for the argument for the Make Call message, and a control object for the same argument.

```
    tMakeCallArgument_T.callingDevice.t = T_DeviceID_dialingNumber;
    tMakeCallArgument_T.callingDevice.u.dialingNumber = pszCallingDevice;
```

Here the method is indicating that the calling device will be specified as a dialing number (an extension number or phone number, in other words), and it specifies what that number is.

```
    ASN1T_DeviceID tDeviceID;
```

```
tDeviceID.t = T_DeviceID_dialingNumber;
tDeviceID.u.dialingNumber = pszCalledDevice;
```

Here the method is setting up the first of two structures that will be needed to specify the number that is being called.

```
ASN1T_ExtendedDeviceID tExtendedDeviceID;
tExtendedDeviceID.t = T_ExtendedDeviceID_deviceIdentifier;
tExtendedDeviceID.u.deviceIdentifier = &tDeviceID;
```

Here the method is setting up the second of the two structures that will be needed to specify the number that is being called.

```
tMakeCallArgument_T.calledDirectoryNumber.t =
T_CalledDeviceID_deviceIdentifier;
tMakeCallArgument_T.calledDirectoryNumber.u.deviceIdentifier =
&tExtendedDeviceID;
```

Here the method is setting up the fields within the actual Make Call argument for the number that is being called. Note that the Make Call argument refers to the extended device id structure, and the extended device id structure refers to the device id structure.

```
int iLength = tMakeCallArgument_C.Encode();
if (iLength < 0) return NULL;
```

Here the method is encoding the Make Call argument and checking to see if the encoding worked.

```
ASN1T_CSTA_ROSE_PDU tROSEHeaderEnc_T;
ASN1C_CSTA_ROSE_PDU tROSEHeaderEnc_C(tEncodeBuffer,
tROSEHeaderEnc_T);
ASN1T_CSTA_ROSE_PDU_invoke tInvokeObject;
```

Every CSTA message is wrapped within a ROSE header, so here the method is allocating the objects and structures needed to encode the header. Note that the control object for the ROSE header is constructed using the same encode buffer instance that was used for the control object for the Make Call argument. The same buffer is used because the ROSE header will wrap; i.e, include, the information in the Make Call argument.

The invoke object is used because the ROSE operation that is used to tell the PBX to make a call is INVOKE.

```
tInvokeObject.m.argumentPresent = 1;
tInvokeObject.invokeId.t = T_InvokeId_present;
tInvokeObject.invokeId.u.present = 1;
tInvokeObject.opcode.t = T_Code_local;
tInvokeObject.opcode.u.local = 10;
```

Here the method is populating the invoke object. The first line indicates that an argument will be present in the ROSE header. This argument will be the encoded Make Call argument.

The second line indicates that an invoke id will be present in the header. The invoke id is any number to identify this instance of the message that the PBX will accept.

The third line sets the invoke id to 1. The invoke id should be different for each CSTA message sent to the PBX, though wrapping back to 1 after some maximum value will likely be necessary. In some cases the invoke id can contain a couple of pieces of information, again depending on what the PBX will accept.

The fourth line indicates that the opcode that will be used is a local opcode; i.e., defined in the CSTA ASN.1 specification, as opposed to a global opcode.

The fifth line sets the opcode to 10, which is the opcode for Make Call.

```
    tInvokeObject.argument.numocts = iLength;
    tInvokeObject.argument.data = (OSOCTET *) tEncodeBuffer.getMsgPtr();
```

Here the method is setting the already-encoded Make Call argument into the invoke object.

```
    tROSEHeaderEnc_T.t = T_CSTA_ROSE_PDU_invoke;
    tROSEHeaderEnc_T.u.invoke = &tInvokeObject;
```

Here the method "tells" the ROSE header that the operation will be an INVOKE operation and sets the invoke object to the header.

```
    iLength = tROSEHeaderEnc_C.Encode();
    if (iLength < 0) return NULL;
```

Here the method encodes the ROSE header and checks to see if the encoding worked. The encoded ROSE header will include the bytes of the already-encoded Make Call argument.

At this point the Make Call argument is completely encoded. So the next step for this method is to send the message to the PBX, using whatever mechanism is necessary (e.g., TCP/IP). We will assume that the message has been sent and a response received. We will further assume that pachResponseFromPBX points to the bytes of the encoded response message, and that iResponseFromPBX contains the length of the response message.

```
    ASN1BERDecodeBuffer tROSEDecodeBuffer(pachResponseFromPBX,
iResponseFromPBX);
    ASN1T_CSTA_ROSE_PDU tROSEHeaderDec_T;
    ASN1C_CSTA_ROSE_PDU tROSEHeaderDec_C(tROSEDecodeBuffer,
tROSEHeaderDec_T);
```

Here the method is allocating some objects to decode the ROSE header of the response. The first line allocates a decode buffer. The second line allocates a data object. And the third line allocates a control object.

```
    int iStatus = tROSEHeaderDec_C.Decode();
    if (iStatus != 0) return NULL;
```

Here the method decodes the ROSE header and checks to see if the decoding worked.

```
    if (tROSEHeaderDec_T.t != T_CSTA_ROSE_PDU_returnResult) return NULL;
    if (tROSEHeaderDec_T.u.returnResult->result.opcode.t !=
T_Code_local) return NULL;
    if (tROSEHeaderDec_T.u.returnResult->result.opcode.u.local != 10)
return NULL;
```

Here the method does a few sanity checks against the decoded ROSE header. The first line checks to make sure that the message is a RETURN RESULT message. The second line checks to make sure that the opcode associated with the response message is a local opcode. And the third line checks to make sure that the opcode associated with the response message is the Make Call opcode.

The method could also check the invoke id of the result message at this point. The invoke id is contained in the invokeId field of returnResult. The invoke id of a result message should be the same as the invoke id of the original INVOKE message.

```
    ASN1BERDecodeBuffer
tContentDecodeBuffer(tROSEHeaderDec_T.u.returnResult-
>result.result.data, tROSEHeaderDec_T.u.returnResult-
>result.result.numocts);
    ASN1T_MakeCallResult tMakeCallResult_T;
    ASN1C_MakeCallResult tMakeCallResult_C(tContentDecodeBuffer,
tMakeCallResult_T);
```

Here the method is allocating objects needed to decode the actual content of the response message, as opposed to the ROSE header. In this case the content is a Make Call Result message.

```
    iStatus = tMakeCallResult_C.Decode();
    if (iStatus != 0) return NULL;
```

Here the method is decoding the contents of the response message and checking to see if the decoding worked.

```
    if (tMakeCallResult_T.t != T_MakeCallResult_initiatedCall) return
NULL;
    ASN1T_ConnectionID *ptConnectionID =
tMakeCallResult_T.u.initiatedCall;
    if (!ptConnectionID->m.callPresent) return NULL;
```

Here the method is checking to make sure the result message contains the needed information about the new call and that the new call's connection id contains the call id.

```
    ASN1TDynOctStr tCallID = ptConnectionID->call;
    char *pachCallID = new char [tCallID.numocts];
    memcpy(pachCallID, tCallID.data, tCallID.numocts);
    return pachCallID;
```

Here the method is extracting the octets of the call id from the response message. A new buffer is allocated on the heap for the octets of the call id so there is no pointer to memory that might go out of scope.

Returning the pointer to the ASN1T_ConnectionID structure itself is also possible. As with the call id, a copy of the connection id should be made and a pointer to that copy returned. Such a copy operation is not as trivial as it may sound, as the ASN1T_ConnectionID structure can have multiple levels of nested structure pointers. ASN1C can generate copy functions for all generated structures and objects by adding -gencopy to its command line (in the makefile or the Visual Studio project). If copy functions are generated, the function to copy an ASN1T_ConnectionID instance can be invoked to do a complete copy of the structure.

Monitoring

A frequent need in CSTA applications is to monitor one or more telephony devices. There are three parts to monitoring: starting a monitor, receiving monitor events, and stopping a monitor.

Starting a Monitor

The code sample below shows how a monitor can be started. The sample uses a method called startMonitor() within a class called CSTAEngine. The method accepts one argument (the extension

number to monitor) and returns a pointer to the monitor cross reference id (a sequence of bytes assigned by the PBX to identify the monitor). If anything goes wrong, the method simply returns NULL.

This code sample uses the following variable prefix conventions:

> psz – Pointer to null-terminated string.
> pach – Pointer to an array of chars (not necessarily null-terminated).
> t – Structure or object instance.
> pt – Pointer to a structure or object instance.
> i – Integer.

If a variable ends with _T, the variable refers to an ASN1C-generated data object. If a variable ends with _C, the variable refers to an ASN1C-generated control object.

```
#include "CSTA-status-reporting.h"
#include "CSTA-monitor-start.h"
#include "CSTA-device-identifiers.h"
#include "CSTA-switching-function-objects.h"
#include "CSTA-ROSE-PDU-types.h"

#include "asn1BerCppTypes.h"
#include "asn1CppTypes.h"
#include "ASN1TOctStr.h"
#include "osSysTypes.h"

#include "CSTAEngine.h"
```

These are the include directives that are necessary.

```
ASN1T_MonitorCrossRefID *CSTAEngine::startMonitor(char
*pszDeviceToMonitor)
{
    ASN1BEREncodeBuffer tEncodeBuffer;
    ASN1T_MonitorStartArgument tMonStartArgument_T;
    ASN1C_MonitorStartArgument tMonStartArgument_C(tEncodeBuffer,
tMonStartArgument_T);
```

In this section the method is allocating an encode buffer, a data object for the argument for the Monitor Start message, and a control object for the same argument.

```
    ASN1T_DeviceID tDevice;
    tDevice.t = T_DeviceID_dialingNumber;
    tDevice.u.dialingNumber = pszDeviceToMonitor;
```

Here the method is setting up a structure that will be needed to specify the number that is being called. The method is indicating that the device to monitor will be specified as a dialing number (an extension number or phone number, in other words), and it specifies what that number is.

```
    tMonStartArgument_T.monitorObject.t = T_CSTAObject_device;
    tMonStartArgument_T.monitorObject.u.device = &tDevice;
```

Here the method is tying the object for the Monitor Start argument's data to the device structure that it just populated.

```
    int iLength = tMonStartArgument_C.Encode();
    if (iLength < 0) return NULL;
```

Here the method is encoding the Monitor Start argument and checking to see if the encoding worked.

```
    ASN1T_CSTA_ROSE_PDU tROSEHeaderEnc_T;
    ASN1C_CSTA_ROSE_PDU tROSEHeaderEnc_C(tEncodeBuffer,
tROSEHeaderEnc_T);
    ASN1T_CSTA_ROSE_PDU_invoke tInvokeObject;
```

Every CSTA message is wrapped within a ROSE header, so here the method is allocating the objects and structures needed to encode the header. Note that the control object for the ROSE header is constructed using the same encode buffer instance that was used for the control object for the Monitor Start argument. The same buffer is used because the ROSE header will wrap; i.e, include, the information in the Monitor Start argument.

```
    tInvokeObject.m.argumentPresent = 1;
    tInvokeObject.invokeId.t = T_InvokeId_present;
    tInvokeObject.invokeId.u.present = 1;
    tInvokeObject.opcode.t = T_Code_local;
    tInvokeObject.opcode.u.local = 71;
```

Here the method is populating the invoke object. The first line indicates that an argument will be present in the ROSE header. This argument will be the encoded Monitor Start argument.

The second line indicates that an invoke id will be present in the header. The invoke id is any number to identify this instance of the message that the PBX will accept.

The third line sets the invoke id to 1. The invoke id should be different for each CSTA message sent to the PBX, though wrapping back to 1 after some maximum value will likely be necessary. In some cases the invoke id can contain a couple of pieces of information, again depending on what the PBX will accept.

The fourth line indicates that the opcode that will be used is a local opcode; i.e., defined in the CSTA ASN.1 specification, as opposed to a global opcode.

The fifth line sets the opcode to 71, which is the opcode for Monitor Start.

```
    tInvokeObject.argument.numocts = iLength;
    tInvokeObject.argument.data = (OSOCTET *) tEncodeBuffer.getMsgPtr();
```

Here the method is setting the already-encoded Monitor Start argument into the invoke object.

```
    tROSEHeaderEnc_T.t = T_CSTA_ROSE_PDU_invoke;
    tROSEHeaderEnc_T.u.invoke = &tInvokeObject;
```

Here the method "tells" the ROSE header that the operation will be an INVOKE operation and sets the invoke object to the header.

```
    iLength = tROSEHeaderEnc_C.Encode();
    if (iLength < 0) return NULL;
```

Here the method encodes the ROSE header and checks to see if the encoding worked. The encoded ROSE header will include the bytes of the already-encoded Monitor Start argument.

At this point the Monitor Start argument is completely encoded. So the next step for this method is to send the message to the PBX, using whatever mechanism is necessary (e.g., TCP/IP). We will assume that the message has been sent and a response received. We will further assume that pachResponseFromPBX

points to the bytes of the encoded response message, and that iResponseFromPBX contains the length of the response message.

```
    ASN1BERDecodeBuffer tROSEDecodeBuffer(pachResponseFromPBX,
iResponseFromPBX);
    ASN1T_CSTA_ROSE_PDU tROSEHeaderDec_T;
    ASN1C_CSTA_ROSE_PDU tROSEHeaderDec_C(tROSEDecodeBuffer,
tROSEHeaderDec_T);
```

Here the method is allocating some objects to decode the ROSE header of the response. The first line allocates a decode buffer. The second line allocates a data object. And the third line allocates a control object.

```
    int iStatus = tROSEHeaderDec_C.Decode();
    if (iStatus != 0) return NULL;
```

Here the method decodes the ROSE header and checks to see if the decoding worked.

```
    if (tROSEHeaderDec_T.t != T_CSTA_ROSE_PDU_returnResult) return NULL;
    if (tROSEHeaderDec_T.u.returnResult->result.opcode.t !=
T_Code_local) return NULL;
    if (tROSEHeaderDec_T.u.returnResult->result.opcode.u.local != 71)
return NULL;
```

Here the method does a few sanity checks against the decoded ROSE header. The first line checks to make sure that the message is a RETURN RESULT message. The second line checks to make sure that the opcode associated with the response message is a local opcode. And the third line checks to make sure that the opcode associated with the response message is the Monitor Start opcode.

The method could also check the invoke id of the result message at this point. The invoke id is contained in the invokeId field of returnResult. The invoke id of a result message should be the same as the invoke id of the original INVOKE message.

```
    ASN1BERDecodeBuffer
tContentDecodeBuffer(tROSEHeaderDec_T.u.returnResult-
>result.result.data, tROSEHeaderDec_T.u.returnResult-
>result.result.numocts);
    ASN1T_MonitorStartResult tMonStartResult_T;
    ASN1C_MonitorStartResult tMonStartResult_C(tContentDecodeBuffer,
tMonStartResult_T);
```

Here the method is allocating objects needed to decode the actual content of the response message, as opposed to the ROSE header. In this case the content is a Monitor Start Result message.

```
    iStatus = tMonStartResult_C.Decode();
    if (iStatus != 0) return NULL;
```

Here the method is decoding the contents of the response message and checking to see if the decoding worked.

```
    ASN1T_MonitorCrossRefID tMonCrossRef =
tMonStartResult_T.crossRefIdentifier;
    ASN1T_MonitorCrossRefID *ptReturnValue = new
ASN1T_MonitorCrossRefID(tMonCrossRef);
    return ptReturnValue;
```

Here the method is extracting the monitor cross reference id from the result message data. The method makes a copy of this cross reference id on the heap and returns a pointer to the copy so the caller won't have a pointer to memory that might go out of scope.

Receiving Monitor Events

When a monitor is established against a device, the PBX sends event report messages to the client software. For example, lifting the receiver of a monitored device usually causes at least one event report message to be sent by the PBX.

The code sample below shows how an event report message can be handled. The sample uses a method called handleEvent() within a class called CSTAEngine. The method accepts two arguments (a pointer to the bytes of an encoded event report message and the length of the message). So the method assumes that the client code has a mechanism set up to receive event report messages from the PBX using a communication mechanism like TCP/IP. For the purposes of this sample any events other than Established events are ignored.

This code sample uses the following variable prefix conventions:

> psz – Pointer to null-terminated string.
> pach – Pointer to an array of chars (not necessarily null-terminated).
> t – Structure or object instance.
> pt – Pointer to a structure or object instance.
> i – Integer.

If a variable ends with _T, the variable refers to an ASN1C-generated data object. If a variable ends with _C, the variable refers to an ASN1C-generated control object.

```
#include "CSTA-status-reporting.h"
#include "CSTA-monitor-start.h"
#include "CSTA-device-identifiers.h"
#include "CSTA-switching-function-objects.h"
#include "CSTA-event-report-definitions.h"
#include "CSTA-established-event.h"

#include "CSTA-ROSE-PDU-types.h"

#include "asn1BerCppTypes.h"
#include "asn1CppTypes.h"
#include "ASN1TOctStr.h"
#include "osSysTypes.h"

#include "CSTAEngine.h"
```

These are the include directives that are necessary. An assumption is being made here that this CSTAEngine class is the same class that has the previously discussed startMonitor() method, so some of these include directives pertain to that method instead of the handleEvent() method.

```
void CSTAEngine::handleEvent(OSOCTET *pachEventMessage, int
iEventMessage)
{
    ASN1BERDecodeBuffer tROSEDecodeBuffer(pachEventMessage,
iEventMessage);
    ASN1T_CSTA_ROSE_PDU tROSEHeaderDec_T;
    ASN1C_CSTA_ROSE_PDU tROSEHeaderDec_C(tROSEDecodeBuffer,
tROSEHeaderDec_T);
```

Here the method is allocating some objects to decode the ROSE header of the event report message. The first line allocates a decode buffer. The second line allocates a data object. And the third line allocates a control object.

```
int iStatus = tROSEHeaderDec_C.Decode();
if (iStatus != 0) return;
```

Here the method decodes the ROSE header and checks to see if the decoding worked.

```
if (tROSEHeaderDec_T.t != T_CSTA_ROSE_PDU_invoke) return;
if (tROSEHeaderDec_T.u.invoke->opcode.t != T_Code_local) return;
if (tROSEHeaderDec_T.u.invoke->opcode.u.local != 21) return;
```

Here the method does a few sanity checks against the decoded ROSE header. The first line checks to make sure that the message is an INVOKE message (CSTA event report messages are INVOKE messages instead of RETURN RESULT messages).

The second line checks to make sure that the opcode associated with the message is a local opcode. Notice that the choice within the ROSE header object's union is the "invoke" member instead of the "returnResult" member.

And the third line checks to make sure that the opcode associated with the response message is the event report message opcode.

Since event report messages are INVOKE messages, the invoke id is most likely not going to be the same as the invoke id of the Monitor Start message that was sent to start the monitor.

```
ASN1BERDecodeBuffer tContentDecodeBuffer(tROSEHeaderDec_T.u.invoke-
>argument.data, tROSEHeaderDec_T.u.invoke->argument.numocts);
ASN1T_CSTAEventReportArgument tEventReportArg_T;
ASN1C_CSTAEventReportArgument
tEventReportArg_C(tContentDecodeBuffer, tEventReportArg_T);
```

Here the method is allocating objects needed to decode the actual content of the message, as opposed to the ROSE header. In this case the content is an Event Report Argument.

```
iStatus = tEventReportArg_C.Decode();
if (iStatus != 0) return;
```

Here the method is decoding the contents of the message and checking to see if the decoding worked.

```
ASN1T_EventSpecificInfo tEventSpecificInfo =
tEventReportArg_T.eventSpecificInfo;
if (tEventSpecificInfo.t != T_EventSpecificInfo_callEvent) return;
```

Here the method is checking to see if the event is one of the events in the group known as call events. The Established event is a call event.

```
ASN1T_CallEvent *ptCallEvent = tEventSpecificInfo.u.callEvent;
if (ptCallEvent->t != T_CallEvent_establishedEvent) return;
ASN1T_EstablishedEvent *ptEstablishedEvent = ptCallEvent-
>u.establishedEvent;
```

Here the method checks to make sure the event is an Established event (line 2). Then the method isolates the information that describes this event.

At this point the variable ptEstablishedEvent points to the information about the Established event, and the method can do whatever it needs to do with that information.

Stopping a Monitor

The code sample below shows how a monitor can be stopped. The sample uses a method called stopMonitor() within a class called CSTAEngine. The method accepts one argument (the cross reference id of the monitor to stop) and returns 0 for success or -1 for failure.

This code sample uses the following variable prefix conventions:

> psz – Pointer to null-terminated string.
> pach – Pointer to an array of chars (not necessarily null-terminated).
> t – Structure or object instance.
> pt – Pointer to a structure or object instance.
> i – Integer.

If a variable ends with _T, the variable refers to an ASN1C-generated data object. If a variable ends with _C, the variable refers to an ASN1C-generated control object.

```
#include "CSTA-status-reporting.h"
#include "CSTA-monitor-start.h"
#include "CSTA-monitor-stop.h"
#include "CSTA-device-identifiers.h"
#include "CSTA-switching-function-objects.h"
#include "CSTA-event-report-definitions.h"
#include "CSTA-established-event.h"

#include "CSTA-ROSE-PDU-types.h"

#include "asn1BerCppTypes.h"
#include "asn1CppTypes.h"
#include "ASN1TOctStr.h"
#include "osSysTypes.h"

#include "CSTAEngine.h"
```

These are the include directives that are necessary.   An assumption is being made here that this CSTAEngine class is the same class that has the previously discussed startMonitor() and handleEvent() methods, so some of these include directives pertain to those methods instead of the stopMonitor() method.

```
int CSTAEngine::stopMonitor(ASN1T_MonitorCrossRefID *ptCrossRefID)
{
    ASN1BEREncodeBuffer tEncodeBuffer;
    ASN1T_MonitorStopArgument tMonStopArgument_T;
    ASN1C_MonitorStopArgument tMonStopArgument_C(tEncodeBuffer,
tMonStopArgument_T);
```

In this section the method is allocating an encode buffer, a data object for the argument for the Monitor Stop message, and a control object for the same argument.

```
    tMonStopArgument_T.t = T_MonitorStopArgument_crossRefIdentifier;
    tMonStopArgument_T.u.crossRefIdentifier = ptCrossRefID;
```

Here the method is identifying the choice within the Monitor Stop argument that is going to be used: the choice to specify a cross reference id. Then it populates the correct member of the choice's union with the pointer to the cross reference id.

```
int iLength = tMonStopArgument_C.Encode();
if (iLength < 0) return -1;
```

Here the method is encoding the Monitor Stop argument and checking to see if the encoding worked.

```
ASN1T_CSTA_ROSE_PDU tROSEHeaderEnc_T;
ASN1C_CSTA_ROSE_PDU tROSEHeaderEnc_C(tEncodeBuffer,
tROSEHeaderEnc_T);
ASN1T_CSTA_ROSE_PDU_invoke tInvokeObject;
```

Every CSTA message is wrapped within a ROSE header, so here the method is allocating the objects and structures needed to encode the header. Note that the control object for the ROSE header is constructed using the same encode buffer instance that was used for the control object for the Monitor Stop argument. The same buffer is used because the ROSE header will wrap; i.e, include, the information in the Monitor Stop argument.

```
tInvokeObject.m.argumentPresent = 1;
tInvokeObject.invokeId.t = T_InvokeId_present;
tInvokeObject.invokeId.u.present = 1;
tInvokeObject.opcode.t = T_Code_local;
tInvokeObject.opcode.u.local = 73;
```

Here the method is populating the invoke object. The first line indicates that an argument will be present in the ROSE header. This argument will be the encoded Monitor Stop argument.

The second line indicates that an invoke id will be present in the header. The invoke id is any number to identify this instance of the message.that the PBX will accept.

The third line sets the invoke id to 1. The invoke id should be different for each CSTA message sent to the PBX, though wrapping back to 1 after some maximum value will likely be necessary. In some cases the invoke id can contain a couple of pieces of information, again depending on what the PBX will accept.

The fourth line indicates that the opcode that will be used is a local opcode; i.e., defined in the CSTA ASN.1 specification, as opposed to a global opcode.

The fifth line sets the opcode to 73, which is the opcode for Monitor Stop.

```
tInvokeObject.argument.numocts = iLength;
tInvokeObject.argument.data = (OSOCTET *) tEncodeBuffer.getMsgPtr();
```

Here the method is setting the already-encoded Monitor Stop argument into the invoke object.

```
tROSEHeaderEnc_T.t = T_CSTA_ROSE_PDU_invoke;
tROSEHeaderEnc_T.u.invoke = &tInvokeObject;
```

Here the method "tells" the ROSE header that the operation will be an INVOKE operation and sets the invoke object to the header.

```
iLength = tROSEHeaderEnc_C.Encode();
if (iLength < 0) return -1;
```

Here the method encodes the ROSE header and checks to see if the encoding worked. The encoded ROSE header will include the bytes of the already-encoded Monitor Stop argument.

At this point the Monitor Stop argument is completely encoded. So the next step for this method is to send the message to the PBX, using whatever mechanism is necessary (e.g., TCP/IP). We will assume that the message has been sent and a response received. We will further assume that pachResponseFromPBX points to the bytes of the encoded response message, and that iResponseFromPBX contains the length of the response message.

```
    ASN1BERDecodeBuffer tROSEDecodeBuffer(pachResponseFromPBX,
iResponseFromPBX);
    ASN1T_CSTA_ROSE_PDU tROSEHeaderDec_T;
    ASN1C_CSTA_ROSE_PDU tROSEHeaderDec_C(tROSEDecodeBuffer,
tROSEHeaderDec_T);
```

Here the method is allocating some objects to decode the ROSE header of the response. The first line allocates a decode buffer. The second line allocates a data object. And the third line allocates a control object.

```
    int iStatus = tROSEHeaderDec_C.Decode();
    if (iStatus != 0) return -1;
```

Here the method decodes the ROSE header and checks to see if the decoding worked.

```
    if (tROSEHeaderDec_T.t != T_CSTA_ROSE_PDU_returnResult) return -1;
    return 0;
```

Here the method checks to make sure that the message is a RETURN RESULT message, which indicates a successful operation. In this case there is no other information in the response message that's needed, so the method then simply returns with success status.

The method could also check the invoke id of the result message at this point. The invoke id is contained in the invokeId field of returnResult. The invoke id of a result message should be the same as the invoke id of the original INVOKE message.