CIL Coding Conventions

This chapter discusses coding conventions used in the CTI OS Client Interface Library (CIL). Coding conventions are standard ways of performing common tasks. While the rest of this document discusses the programming interfaces available with the CIL, this chapter provides useful and practical explanation of how to program with the CIL– the glue that brings everything together.

One of the design goals of the CTI OS CIL is to make programming as easy and consistent as possible for client developers. As such, many design decisions about the CIL interfaces were made to keep things simple, clear, and consistent across various objects, methods, and programming environments.

This chapter discusses the following topics:

- Data types
- Asynchronous execution (error codes versus events)
- Generic interfaces with the Arguments structure
- Optional and reserved parameters
- Accessing properties and parameters with GetValue
- Adding parameters to requests with AddItem
- Setting properties with SetValue
- UniqueObjectIDs: how to identify objects
- Obtaining an object from its UniqueObjectID
- Using Button Enablement Masks
- Methods that call AddRef()

- CTI OS CIL Data Types, page 2
- Asynchronous Program Execution, page 3
- CIL Error Codes, page 3
- COM Error Codes, page 8
- Generic Interfaces, page 9
- UniqueObjectID Variable-Length String, page 11

- UniqueObjectID to Obtain Pointer or Reference, page 12
- Button Enablement Masks, page 13

CTI OS CIL Data Types

The CTI OS Client Interface Library is designed to be a single interface, which you can use across multiple languages and environments (e.g. C++, COM, Visual Basic, Java, and .NET). However, each programming language has its own native data types. Throughout this document, the interface parameters are listed with the following standardized data types:

- **STRING**: A variable-length string variable. If a maximum length exists, it is listed with the parameter description.
- **INT**: A *32-bit* wide integer.
- **UNSIGNED INT**: A *32-bit* wide unsigned integer.
- **SHORT**: A *16-bit* wide short integer.
- **UNSIGNED SHORT**: A *16-bit* wide unsigned short integer.
- **BOOL**: A logical *true* or *false* variable. Different implementations use variables of different sizes to represent this type. In COM, the VARIANT_BOOL is used. Tests of variables of this data type must be against VARIANT_TRUE and VARIANT_FALSE and not simply against 0 or 1.
- **ARGUMENTS**: A custom data structure used by CTI OS, which holds a variable-length set of key-value pairs.
- ARG: An individual element (value), which can be stored in an ARGUMENTS structure.

The following table describes the appropriate language specific types to which the documented type are associated.

Table 1: CTI OS CIL Data Type

Documented Data Type	STRING	INT	UNSIGNED INT	SHORT	UNSIGNED SHORT	BOOL	ARGUMENTS	ARG
C++ Type	std::string or const char	long or int	unsigned int	short	unsigned short	bool	Arguments	Arg
Visual Basic 6.0 Type	String	Long	None	Integer	Integer	Boolean	Arguments	Arg
COM Type	BSTR	long or int	unsigned int	short	unsigned short	VARIANT_BOOL	IArguments *	IArg*
Java Type	String	int	long	short	int	Boolean	Arguments	Arg
.NET Type	System.String	System.Int32	System.Int64	System.Int16	System.Int32	System.Boolean	Arguments	Arg

Asynchronous Program Execution

The most common programming approach used by applications is synchronous execution. In a synchronous execution mode, a method call executes all the code required to complete the request and provide return values as well as error codes. Client-server programming can be synchronous (the client application makes a blocking request and continues execution when the request is completed) or asynchronous (the client application makes a request and continues processing immediately, with the result of the request to follow at a later time).

CTI programming is unique in that requests are often serviced by third-party servers or applications, such as a PBX/ACD in the contact center. The asynchronous nature of CTI programming requires developers to note the distinction between an error code and the response to a request. In non-CTI programming, developers test the error codes (return values from method calls) to determine whether a method request succeeded or failed. However, in a distributed architecture such as CTI OS, success or failure is often determined by some external server or component such as the PBX/ACD.

The CTI OS Client Interface Library API specifies error codes, which are return values for method calls. These error codes relate to the success or failure of the method call, but not the success or failure of the underlying operation. The success of the method call means that the parameters sent were of the correct format, that internal memory allocations were successful, and that the request was put on the send queue to be transmitted to the CTI OS Server. Generally, the CIL error code returned from method calls is CIL_OK, indicating that the method call was successful. However, this does not indicate that the request was actually serviced by the CTI OS Server or successfully completed at the PBX/ACD.

To determine the success or failure of the underlying telephony operation requested, the CTI programmer must wait for an event confirming the success or failure of the request. To generalize the message flow model, most requests made at the CTI OS CIL are answered with a confirmation message and/or an event message. See the object interface reference in Chapters 8-12 for details on each particular request. This type of response is called asynchronous—it can arrive at any time after the request is made, but typically requests are services in sub-second timeframes.

The expected event sequence is described for each method request in the programmer's interface sections of this document so that programmers know which events to expect. In the event of a request failure, an eControlFailureConf message is sent to the client; the eControlFailureConf message has a parameter called MessageType indicating which request failed, and a parameter called ErrorMessage, with a description of the failure cause.

For example, when sending a MakeCall request, the method typically returns CIL_OK, which means that the method call was successful. If the underlying make call request is successful, the CIL receives several follow-on events, such as eBeginCallEvent and eServiceInitiatedEvent. If the request fails, the CIL receives the eControlFailureConf message.

A common mistake is that developers who have not previously programmed with asynchronous events mistake the error code returned from a method call for the actual result of the request. The correct semantics are to interpret the error code as being indicative of the result of the method call, and to interpret the follow-on events to determine the actual result of the requested operation.

CIL Error Codes

Whenever a method call is invoked by a custom application using the CIL, an error code is returned. The error codes returned only indicate success or failure of the method call, as indicated in the previous section.

The possible values of the error code returned from C++ and Java CIL methods are defined in the following table.

Note The numeric values listed in the following table are subject to change. Use the error code enumerations to check a given error code, rather than rely on a specific numeric value.

Table 2: CIL Error Codes

CIL Error Code	Numeric Value	Description
CIL_OK	1	The method succeeded. The request to silent monitor the call was successfully initiated.
CIL_FAIL	0	The method failed.
E_CTIOS_METHOD_NO_ IMPLEMENTED	-99	There is no implementation available for this method.
E_CTIOS_INVALID_ PROPERTY	-100	One or more properties are invalid.
E_CTIOS_MODE_CONFLICT	-101	A conflict when setting session mode.
E_CTIOS_INVALID_ EVENTID	-102	The Event ID is not valid.
E_CTIOS_INVALID_ ARGUMENT	-103	The Argument is not valid.
E_CTIOS_INVALID_ SESSION	-104	The Session is not valid.
E_CTIOS_UNEXPECTED	-105	An unexpected error has occurred.
E_CTIOS_OBJ_ALLOCATION_FAILED	-106	There is not enough memory available and the creation of CCtiOsObject failed.
E_CTIOS_ARRAYREF_ ALLOCATION_FAILED	-107	There is not enough memory available and the creation of an array of references to objects of type CCtiOsObject failed.
E_CTIOS_ARGUMENT_ ALLOCATION_FAILED	-108	There is not enough memory available and the creation of an object of type Arguments failed.
E_CTIOS_TARGET_ OBJECT_ NOT_FOUND	-109	There are no CTI OS Objects capable of processing an incoming event.
E_CTIOS_PROP_ ATTRIBUTES_ACCESS_ FAILED	-110	An error occurred while accessing a property's attributes, System may be running out of memory.
E_CTIOS_INVALID_ OBJECT_TYPE	-111	The object type is not one of the following predefined types CAgent, CCall, CSkillGroups, or CWaitObject.
E_CTIOS_INVALID_AGENT	-112	No valid agent.

CIL Error Code	Numeric Value	Description
E_CTIOS_INVALID_CALL	-113	No valid call.
E_CTIOS_IN_FAILOVER	-114	The session is recovering from a connection failure and started the Fail Over procedure.
E_CTIOS_INVALID_ DESKTOP_TYPE	-115	Indicates that the desktop type specified in the request for DeskSettings download is neither Agent or Supervisor.
E_CTIOS_MISSING_ ARGUMENT	-116	Missing a required argument.
E_CTIOS_CALL_NOT_ON_ HOLD	-117	Call is not on hold.
E_CTIOS_CALL_ALREADY_ ON_HOLD	-118	Call is already on hold.
E_CTIOS_CALL_NOT_ ALERTING	-119	Call is not in alert state, it can not be answered.
E_CTIOS_AGENT_NOT_ LOGIN	-120	Agent is not logged in.
E_CTIOS_INVALID_ METHOD_PARAMETER	-121	The input parameter is invalid.
E_CTIOS_UNKNOWN	-122	The cause of this error is unknown.
E_CTIOS_OUT_OF_ MEMORY	-123	Failed to allocate new memory.
E_CTIOS_PORT_ UNAVAILABLE	-124	The specified port is not available for use.
E_CTIOS_SM_SESSION_ TERMINATED_ ABNORMALLY	-125	The Silent Monitor session was terminated abnormally.
E_CTIOS_SM_REJECTED_ ALREADY_IN_SESSION	-126	The request was rejected because there is an active silent monitor session in progress.
E_CTIOS_SM_PACKET_ SNIFFER_NOT_INSTALLED	-127	The packet sniffer is not present in the system; verify installation.
E_CTIOS_PACKET_ SNIFFER_FAILED	-128	An error occurred in the packet sniffer.
E_CTIOS_SOCKET_CALL_ FAILED	-129	A CTI OS socket call failed.
E_CTIOS_MEDIA_ TERMINATION_NOT_ INSTALLED	-130	EVVBU Media Termination component in the system, verify installation.
E_CTIOS_MT_UNKNOWN_ CODEC	-131	Specified CODEC is not supported.

CIL Error Code	Numeric Value	Description
E_CTIOS_MEDIA_ TERMINATION_FAILED	-132	An error occurred in the Media Termination Packet Decoder.
E_CTIOS_SNIFFER_NO_ PACKETS_RECEIVED	-133	The Sniffer has not received any IP packets.
E_CTIOS_SNIFFER_ FAILED_TO_OPEN_DEVICE	-134	The Sniffer failed to open the networking device.
E_CTIOS_SNIFFER_ FAILED_TO_SET_FILTER	-135	The Sniffer failed when setting the packet filter.
E_CTIOS_ERROR_IN_ PACKET_FILTER	-136	The packet filter expression is incorrect.
E_CTIOS_INVALID_ MONITORED_IP_ADDRESS	-137	The IP Address specified for the monitored device (IP Phone) is not valid.
E_CTIOS_INVALID_ SNIFFER_OBJECT	-138	Invalid Sniffer object.
E_CTIOS_INVALID_ DECODER_OBJECT	-139	Invalid Decoder object.
E_CTIOS_NO_SM_ SESSION_IN_PROGRESS	-140	There are no Silent Monitor Sessions in progress.
E_CTIOS_INVALID_ SILENT_MONITOR_ SESSION	-141	The specified Silent Monitor session does not exist.
E_CTIOS_FAILED_ REMOVING_SILENT_ MONITOR_SESSION	-142	Silent Monitor Session was not removed from the collection.
E_CTIOS_IP_PHONE_ INFORMATION_NOT_ AVAILABLE	-143	There is no information available about the IP Phone.
E_CTIOS_PEER_NOT_ ENABLED_FOR_SILENT_ MONITOR	-144	The peer application is not enabled for Silent Monitor.
E_CTIOS_NOT_ENABLED_ FOR_SILENT_MONITOR	-145	This application is not enabled for Silent Monitor.
E_CTIOS_NO_PENDING_REQUEST	-146	There are no pending requests to be processed.
E_CTIOS_ALREADY_IN_SESSION	-147	There is already an established session.
E_CTIOS_MODE_SET_ALREADY	-148	The session mode is already set.

CIL Error Code	Numeric Value	Description
E_CTIOS_MODE_NOT_SET	-149	The session mode is not set yet.
E_CTIOS_INVALID_OBJECT_STATE	-150	The object is not in the correct state.
E_CTIOS_INVALID_SILENT_MONITOR_MODE	-151	This error occurs when a request to initiate CTI OS silent monitor is made and CTI OS is configured to use CCM silent monitor. This error also occurs when a request to initiate CCM silent monitor is made and CTI OS is configured to use CTI OS silent monitor.
E_CTIOS_COM_OBJ_ ALLOCATION_FAILED	-200	CoCreateInstance failed to create a COM object wrapper for a CIL Object (Session, Agent, Call, Skill, etc.).
E_CTIOS_COM_ CORRUPTED_REGISTRY	-201	A COM component failed to access data from the registry.
E_CTIOS_COM_DIALPAD_ FAIL_TO_LOAD	-202	The Dial Pad common dialog was not created and CoCreateInstance failed.
E_CTIOS_COM_CONV_ COMPTR_TO_CPPPTR_ FAILED	-203	Failed converting COM pointer to C++ pointer.
E_CTIOS_COM_NOT_ INITIALIZED	-204	The MS COM library is not initialized. Invoke CoInitialize(...).
E_CTIOS_SESSION_ DISCONNECT_PENDING	-300	A disconnect is already pending.
E_CTIOS_SESSION_NOT_ CONNECTED	-301	The session is not connected.
E_CTIOS_SESSION_NOT_DISCONNECTED	-351	The call to Connect failed because the session is not in a disconnected state. The session may be connected or a previous call to Disconnect may not yet be complete.
E_CTIOS_AGENT_ ALREADY_IN_SESSION	-900	An object for this agent already exists in the session.
E_CTIOS_SET_AGENT_ SESSION_DISCONNECT_ REQUIRED	-901	Session must be disconnected before operation.
E_CTIOS_SERVICE_SEND_MESSAGE_FAILED	-902	Could not send message. Session may not be connected.
E_CTIOS_CALL_ALREADY_CURRENT_IN_SESSION	-903	An object for this call is already set as current in the session.

CIL Error Code	Numeric Value	Description
E_CTIOS_LOGIN_ INCONSISTENT_ ARGUMENTS	-904	The AgentID and/or PeripheralID provided to a Login call do not match the properties set on the Agent object when SetAgent() was called.

Note If a method that is supposed to trigger an event returns an error code, check this return value before continuing to wait for events. Depending on the error code, the event you were waiting for may not be triggered.

COM Error Codes

For applications using the CTI OS CIL for COM, the Microsoft COM layer adds a level of error detection and provides additional error codes, called HRESULTs. For COM method calls in C++, the HRESULT is returned from the method call, and indicates success or failure of the method call. The CIL error code is also returned, but as an [*out*, *retval*] parameter. For example:

```
// COM Example in C++
int errorCode = 0;
HRESULT hr = pCall->Answer(&errorCode);
if (errorCode=CIL_FAILED)
           printf("An error has occurred while answering the call.")
```

In Visual Basic, HRESULT values are hidden under the covers. When an error occurs, a Visual Basic exception is thrown, which can be caught using the On Error: construct. The CIL error code is returned as the result of the method call:

```
 ' VB example:
On Error GoTo Error_handler
Dim errorCode as Long

ErrorCode = pCall.Answer
If ErrorCode = CIL_FAILED
Debug.print "An error has occurred."
```

The complete set of HRESULT values is defined by Microsoft in the header file *winerror.h*. The most common HRESULT values that CTI OS developers see are listed in the following table:

Table 3: COM Error Codes

COM Error Code	Numeric Value	Description
S_OK	0x00000000	The method succeeded.
S_FALSE	0x00000001	The method succeeded, but something unusual happened.

COM Error Code	Numeric Value	Description
E_FAILED	0x80000008	The method failed.
REG_DB_E_ CLASSNOTREG	0x80040143	The class was not found in the registry. You must run regsvr32.exe on the DLL file to register it.

Generic Interfaces

One of the main design goals of CTI OS was to enable future enhancements to the CTI OS feature set without breaking existing interfaces. To accomplish this, a parameter for almost every method and event is an Arguments array containing the actual parameters needed. Therefore, parameters can be added or deleted in future versions without affecting the signature of the method or event. This provides the benefit to developers that code developed to work with one version of the CTI OS developer toolkit works with future versions without requiring any code changes on the client side (except to take advantage of new features). For example, CTI OS automatically sends a new parameter in the Arguments array for an event, without requiring an interface or library code change. The dilemma of creating a generic interface is solved by using generic mechanisms to send parameters with events and request, and to access properties.

Arguments

The CTI OS developer's toolkit makes extensive use of a new data structure (class) called Arguments. Arguments is a structure of key-value pairs that supports a variable number of parameters and accepts any user-defined parameter names. For any given event, the Arguments structure allows the CTI OS Server to send the CIL any new parameters without requiring client side changes. Similarly, for any request, the programmer can send any new parameters without any changes to the underlying layers.

Example of using Arguments in a Visual Basic MakeCall request:

```
Dim args As New Arguments
args.AddItem "DialedNumber", dialthis.Text

If Not 0 = Len(callvar1.Text) Then
' set callvar1
args.AddItem "CallVariable1", callvar1.Text
End If

' send makecall request
m_Agent.MakeCall args, errorcode
```

Java example:

```
Arguments args = new Arguments();
args.SetValue(CtiOs_IkeywordIDs.CTIOS_DIALEDNUMBER, "12345");
args.SetValue(CtiOs_IkeywordIDs.CTIOS_CALLVARIABLE1, "MyData");
int iRet = m_Agent.MakeCall(args);
```

The Arguments structure can store and retrieve all native C/C++, Visual Basic, and .NET, and Java types, as well as nested Arguments structures.

GetValue Method to Access Properties and Parameters

CTI OS makes extensive use of generic data abstraction. The CTI OS CIL objects, as well as the Arguments structure, store all data by key-value pair. Properties and data values in CTI OS are accessible through a generic mechanism called GetValue. For a list of the different GetValue methods, see CtiOs Object or Helper Classes The GetValue mechanism provides for the retrieval of any data element based on its name. This enables the future enhancement of the data set provided for event parameters and object properties without requiring any interface changes to support new parameters or properties. GetValue supports use of string keywords, as shown in the following examples:

```
// C++string sAgentID;
args.GetValueString("AgentID", &sAgentID);

`Visual Basic
Dim sAgentID As String
sAgentID = args.GetValueString "AgentID"

//Java
String  sID     = args.GetValueString(CtiOs_IkeywordIDs.CTIOS_AGENTID);
Integer IPeriph =
args.GetValueIntObj(CtiOs_IkeywordIDs.CTIOS_PERIPHERALID);

if (IPeriph == null)
// Error accessing Peripheral ID! Handle Error here
else
   iPeriph = IPeriph.intValue();
```

CTI OS defines a set of well-known keywords for event parameters and properties. The well-known keywords are of type string and are listed throughout this document with the methods and events for which they are valid. The complete set of valid keywords are listed in the C++ header file, *ctioskeywords.h*, and are provided in the COM (Visual Basic) type library as well. Java CIL keywords are listed in the Javadoc in the description of the CtiOs_IKeywordIDs interface.

SetValue Method to Set Object Properties and Request Parameters

The CIL also provides an extensible mechanism to set properties on CTI OS Client Interface Objects. The SetValue mechanism, available on the CIL Interface Objects (as well as the CTI OS Arguments class), enables setting properties of any known type to the object as a key-value pair.

SetValue, similar to GetValue and AddItem, supports string keywords and enumerated names:

```
// C++
Agent a;
a.SetValue("AgentID", "22866");
a.SetValue(CTIOS_AGENTID, "22866"); // alternative
a.SetValue(ekwAgentID, "22866"); // alternative

`Visual Basic
Dim a As Agent
a.SetValue "AgentID", "22866"

//Java. Note use of the CTIOS_AGENTID version of keywords.
String  sAgentID = "22866";
```

```
Args.SetValue("AgentID", sAgentID);
Args.SetValue(CtiOs_IkeywordIDs.CTIOS_AGENTID, sAgentID);  // alternative
Args.SetValue(ekwAgentID, sAgentID);
```

The complete syntax and usage of the GetValue, AddItem, and SetValue methods is detailed in CtiOs Object The Arguments structure is detailed in Helper Classes

UniqueObjectID Variable-Length String

The CTI OS Server creates and manages the CTI OS objects, representing all interactions for the contact center. The CTI OS Server and CIL use the UniqueObjectID field to match up a CTI OS object on the CIL with the corresponding object on the Server.

The UniqueObjectID is a variable-length string that can uniquely identify the object within the current context of the CTI OS Server and the Unified ICME and CTI Interlink Advanced. The UniqueObjectID comprises an object type (for example, call, agent, skillgroup, and so on), and two or more additional identifying fields. The following table explains the composition of the UniqueObjectID.

Table 4: UniqueObjectID Components

Object Type	Sample UniqueObjectID	Explanation
Call Object	call.5000.202.23901	The Call object is uniquely identified by its PeripheralID (5000, generated by Unified ICM), ConnectionCallID (202, generated by the PBX/ACD), and its ConnectionDeviceID (23901, generated by the PBX/ACD).
Agent Object	agent.5000.22866	The Agent object is uniquely identified by its PeripheralID (5000, generated by Unified ICM), and its agent ID.
Device Object (for events only; no CIL object)	device.5000.23901	The device object is uniquely identified by its PeripheralID (5000, generated by Unified ICM), and its instrument number (configured by the PBX/ACD).
SkillGroup Object	skillgroup.5000.77	The skill group object is uniquely identified by its PeripheralID (5000, generated by Unified ICM), and its SkillGroupNumber (configured by the PBX/ACD).
Team Object (for events only; no CIL object)	team.5000.5001	The team object is uniquely identified by its PeripheralID (5000, generated by Unified ICM), and its TeamID (5001, also generated by Unified ICM).

Note The CTI OS UniqueObjectID is not the same as the Unified ICM globally unique 64 bit key used in the Unified ICME historical databases (called the ICMEnterpriseUniqueID), which exists only for calls. The ICMEnterpriseUniqueID stays with the call even when the call is transferred between call center sites, while the UniqueObjectID for a call is specific to its site (by PeripheralID, ConnectionCallID, and ConnectionDeviceID).

The ICMEnterpriseUniqueID in CTI OS is a variable-length string with the form

```
"icm.routercallkeyday.routercallkeycallid"
```

where routercallkeyday is the field Day in the Unified ICM Route_Call_Detail and Termination_Call_Detail tables, and routercallkeycallid is the field RouterCallKey in the Unified ICM Route_Call_Detail and Termination_Call_Detail tables.

The CTI OS server enables certain types of monitor mode applications that track the pre-call notification event (eTranslationRouteEvent or eAgentPrecallEvent) and seeks to match the call data with the arrival of an eCallBeginEvent.

To do so, the application receives the pre-call notification for calls routed by Unified ICM, (either pre-route, post-route, or translation route), and creates a record (object) using the ICMEnterpriseUniqueID field as the unique key. Later, when the call arrives at the ACD, and is queued or targeted (by the ACD) for a specific agent, the application can match the saved record (object) with the incoming call by the ICMEnterpriseUniqueID field. The following events contain the ICMEnterpriseUniqueID that can associate a call with the saved call information:

- eCallBeginEvent
- eCallDataUpdateEvent
- eSnapshotCallConf
- eCallEndEvent

UniqueObjectID to Obtain Pointer or Reference

Client applications written to take advantage of the CIL can use the UniqueObjectID to obtain a pointer (in C++ or COM for C++) or a reference (in other languages) to the underlying object.

The CIL Session object provides easy access to the object collections via several methods, including GetObjectFromObjectID. GetObjectFromObjectID takes as a parameter the string UniqueObjectID of the desired object, and returns a pointer to the object. Because this mechanism is generic and does not contain specific information about the object type retrieved, the pointer (or reference) returned is a pointer or reference to the base class: a CCtiosObject* in C++, an Object in Visual Basic, an IDispatch* in COM for C++, or CtiOsObject in .NET and Java.

Note The GetObjectFromObjectID method performs an AddRef() on the pointer before it is returned to the programmer.

C++ example:

```
string sUniqueObjectID = "call.5000.101.23901";
Ccall * pCall = NULL;
m_pSession->GetObjectFromObjectID(sUniqueObjectID,
                          (CCtiOsObject**)&pCall);

pCall->Clear();
pCall->Release(); // release our reference to this object
pCall = NULL;
```

Java example:

```
String  sUID = "call.5000.101.23901";
Call rCall = (Call) m_Session.GetObjectFromObjectID(sUID);
```

Button Enablement Masks

The CTI OS Server provides a rich object-level interface to the CTI interactions of the contact center. One of the features the CTI OS Server provides is to evaluate all of the telephony events, and map them to the features permitted by the Cisco CallManager implementation. The CTI OS Server provides a peripheral-independent mechanism for clients to determine which requests are valid at any given time by using a bitmask to indicate which requests are permitted.

For example, the only valid time to answer a call is when the ENABLE_ANSWER bit in the enablement mask is set to the on position. The following C++ example depicts this case:

```
void EventSink::OnCallDeliveredEvent(Arguments& args)
{
           unsigned int unBitMask = 0;
           if (args.IsValid("EnablementMask"))
           {
                   args.GetValueInt("EnablementMask", & unBitMask)
                //do bitwise comparison
                If(unBitMask & ENABLE_ANSWER)
                   m_AnswerButton.Enable();
           }
}
```

Visual Basic.NET Example

```
Private Sub m_session_OnAgentStateChange(ByVal pIArguments As
Cisco.CTIOSCLIENTLib.Arguments) Handles m_session.OnAgentStateChange
        Dim bitmask As Integer

        'Determine the agent's button enablement and update the buttons
on the form
         bitmask = m_Agent.GetValueInt("EnablementMask")

        btnReady.Enabled = False
        btnNotReady.Enabled = False
        btnLogout.Enabled = False
        btnStartMonitoring.Enabled = False

        If bitmask And
```

```
Cisco.CTIOSCLIENTLib.enumCTIOS_EnablementMasks.ENABLE_READY Then
            btnReady.Enabled = True
        End If
        If bitmask And
Cisco.CTIOSCLIENTLib.enumCTIOS_EnablementMasks.ENABLE_NOTREADY Then
            btnNotReady.Enabled = True
        End If
        If bitmask And
Cisco.CTIOSCLIENTLib.enumCTIOS_EnablementMasks.ENABLE_NOTREADY_WITH_REASON
 Then
            btnNotReady.Enabled = True
        End If
        If bitmask And
Cisco.CTIOSCLIENTLib.enumCTIOS_EnablementMasks.ENABLE_LOGOUT Then
            btnLogout.Enabled = True
        End If
        If bitmask And
Cisco.CTIOSCLIENTLib.enumCTIOS_EnablementMasks.ENABLE_LOGOUT_WITH_REASON
 Then
            btnLogout.Enabled = True
        End If

    End Sub
```

The advantage of using this approach is that all of the peripheral-specific details of enabling and disabling buttons is determined in a central location—at the CTI OS Server. This allows future new features to be enabled, and software bugs to be corrected in a central location, which is a great benefit for deploying future releases.

Note

- You must use the button enablement mask generated by CTI OS Server in all cases where Cisco provides button enablement masks. This prevents application impact if changes are made to the event flow.
- Cisco makes no guarantees that the event flow will remain consistent across versions of software.

Warning **The button enablement feature is intended for use in agent mode applications and not for monitor mode applications.**

For any given event, the CTI OS Server calculates the appropriate button enablement bitmask and sends it to the CIL with the event parameters. The button enablement bit masks are discussed in detail in Event Interfaces and Events You can use these masks to write a custom softphone-type application without writing custom code to enable and disable buttons. This approach is also used internally for the CTI OS ActiveX softphone controls.